Jun Hu

胡 军

Design of a Distributed Architecture for Enriching Media Experience in Home Theaters

J.F. Schouten School for
User-System Interaction Research

Design of a Distributed Architecture for Enriching Media Experience in Home Theaters

PROEFONTWERP

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr.ir. C.J. van Duijn, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 30 augustus 2006 om 16.00 uur

door

Jun Hu

geboren te Jiangsu, China

De documentatie van het proefontwerp is goedgekeurd door de promotoren:

prof.dr.ir. L.M.G. Feijs
en
prof.dr. E.H.L. Aarts

Copromotor:
dr.ir. M.D. Janse

Typeset with $\LaTeX\,2_\varepsilon$
Cover design by Christoph Bartneck and Jun Hu
Printed by Printservice Technische Universiteit Eindhoven

ISBN-10: 90-386-2768-8
ISBN-13: 978-90-386-2768-7

A catalogue record is available from the Library Eindhoven University of Technology

To Wenwen and Xiaoxiao

Acknowledgements

This PhD is the result of a process to which many people contributed. First of all I am really grateful to my promotor, prof.dr. Emile Aarts, Vice President of Philips Research, for allowing me to share his vision of Ambient Intelligence and providing me with opportunities to conduct a big part of this work at the Media Interaction group of Philips Research Eindhoven, at the time he was the group leader. Discussions with him had been always intriguing and inspiring. He stretched the horizons of my ambitions but at the same time helped me to keep a clear focus.

My special appreciation goes to my co-promotor, dr.ir. Maddy Janse from Philips Research. Some years ago she introduced me to the 2-year postmaster program User-system Interaction and since then, she has been always encouraging and supportive during my postmaster study and later the PhD project. As the project leader of two European projects (NexTV and ICE-CREAM), she has given me a lot of opportunities and a great deal of freedom in these projects, allowing me to focus on my own research and design interests.

I would like to thank prof.dr.ir. Loe Feijs for being my promotor, my supervisor, and best of all, a friend. Loe not only guided me through the successive stages of this project, but also participated in several parts of the research. I would also thank him for his faith in me during the entire project and for his quick and careful reading during the difficult phases of writing. Every thesis has its errors in the documentation process and my first drafts had a number of mistakes; but thanks to his patient work of proofreading, these mistakes have been kept to a minimum.

I would like to thank the reading committee very much for going through the pages, and for their insightful comments and criticisms. The reading committee consisted of prof.dr.ir. Berry Eggen and prof.dr. Yibin Hou, in addition to my promotors. I am also grateful to prof.dr.ir. Jeu Schouten, prof.dr. Kees Overbeeke, prof.dr. Matthias Rauterberg and prof.dr. Don Bouwhuis for kindly accepting the positions in the doctorate committee, and also for their support in the course of this project. Apart from the committee, Richard Appleby and Yuechen Qian read through some chapters or sections and gave me insightful comments or helped me to improve the text. Loe translated the summary into samenvatting; Joep Frens, Philip Ross and

Rian van Herk helped to improve the translation. Joep did also the last-minute layout check to remove bad page breaks and ugly hyphenations right before I sent everything to the printer.

I have been always enjoying my coffee, my lunch, and of course my research and design together with my colleague at Designed Intelligence group since I joined. In the group are René Ahn, Christoph Bartneck, Loe Feijs, Joep Frens, Kees Overbeeke, Matthias Rauterberg, Philip Ross, Yuechen Qian, Peter Peters and Stephan Wensveen. Besides the other things, Christoph helped to build the Lego robots and to SPSS the experiment results, and also kindly proposed and drafted the tangram cover design for this thesis; Yuechen moved to Philips Research but his comments on the early versions of the formal specifications were very helpful; Joep and Stephan shared with me a lot of sympathy in getting the PhD done; Kees often came to my door and smiled with questions like "finished yet?" to ensure my diligence never waned.

I would also express my gratitude to the secretaries to the Designed Intelligence group, Helen Maas-Zaan and Rian van Herk, for all the many things they helped me with, which are simply too numerous to mention.

As mentioned in this thesis, some parts of this work were done together with Magdalena Bukowska (NexTV user study), Hyun-joo Kong (ICE-CREAM user evaluation) and Christoph Bartneck (TheInterview experiment). The collaboration was very pleasant and beneficial for me and I hope they enjoyed it too.

Finally, and most importantly, great thanks are due to my family: my wife Wen (徐雯), for her love, and her unconditional and continual support at every level that can only be given, not expected; my son, Xiaoxiao (Mark, 胡水), for sparing me a lot of time by doing great at school and managing himself at home; my parents Jinlin Hu (胡金林) and Shushan Chen (陈淑珊), my parents-in-law Dong Xu (徐东) and Xiao Wu (吴晓), my brother Ping Hu (胡平) and my sisters-in-law Binbin (徐彬彬) and Duoduo (徐朵朵), for their help and support. 感谢你们多年来的支持和帮助。

Jun Hu

[illegible]

July 2006, Eindhoven

Contents

Acknowledgements v

List of Figures xvii

List of Tables xxi

1 Introduction 1
- 1.1 Focus of this thesis . . . 4
- 1.2 Approach . . . 5
  - 1.2.1 Spiral model . . . 5
  - 1.2.2 Architecture-based design . . . 6
- 1.3 Outline of this thesis . . . 7

I Requirements and Concepts 9

2 Explorations and Requirements 11
- 2.1 NexTV and TOONS . . . 11
  - 2.1.1 Getting the user input . . . 12
  - 2.1.2 Requirements of TOONS . . . 14
  - 2.1.3 Three versions of TOONS . . . 18
- 2.2 ICE-CREAM and DeepSea . . . 19
  - 2.2.1 DeepSea scenario . . . 20
  - 2.2.2 Functional requirements . . . 22
  - 2.2.3 Levels of interaction . . . 22
- 2.3 Requirements after DeepSea . . . 23
- 2.4 Rapid Robotic Prototyping . . . 24
- 2.5 Concluding remarks . . . 24

3 **StoryML** 25
3.1 Conceptual model of TOONS . . . 25
3.2 When technologies meet the requirements . . . 26
3.2.1 Open standards for synchronized multimedia . . . 26
3.2.2 Architectures for interactive television . . . 27
3.3 Design of StoryML . . . 28
3.3.1 Why XML . . . 29
3.3.2 Simplified model of interactive story . . . 29
3.3.3 Environment and actors . . . 30
3.3.4 Media Objects . . . 30
3.3.5 Timeline . . . 32
3.4 Implementation of StoryML Player . . . 32
3.4.1 PAC based architecture . . . 32
3.4.2 Extending PAC for timed media . . . 33
3.4.3 Architecture of the StoryML Player . . . 35
3.4.4 Media and Interaction Synchronization . . . 37
3.5 Lessons learnt from StoryML . . . 37
3.5.1 Pros . . . 37
3.5.2 Cons . . . 38

4 **Interactive Play Markup Language** 39
4.1 Play . . . 39
4.1.1 Timing in a play . . . 40
4.1.2 Mapping: Assigning roles to actors . . . 40
4.1.3 Interactive play . . . 42
4.2 SMIL . . . 43
4.3 IPML . . . 45
4.3.1 Actor . . . 46
4.3.2 Action . . . 48
4.3.3 Event . . . 48
4.4 Conclude with Alice in Wonderland . . . 49

II Architecture Design 51

5 **Pattern Oriented Architecture Design** 53
5.1 Design Patterns . . . 53
5.1.1 Software design patterns . . . 54
5.1.2 Pattern language . . . 55
5.1.3 Observer pattern in pattern language . . . 55
5.2 Patterns as architectural constructs . . . 59
5.3 Formal specification . . . 60
5.3.1 Problems of informal specifications . . . 60
5.3.2 Observer pattern in Object-Z . . . 61
5.3.3 Benefits of formal specification . . . 64

6 Actions **65**
6.1 Timed Action . . . 65
6.1.1 Intent . . . 65
6.1.2 Context and forces . . . 65
6.1.3 Solution . . . 67
6.1.4 Structure . . . 67
6.1.5 Consequences . . . 67
6.2 Synchronizable Object . . . 69
6.2.1 Intent . . . 69
6.2.2 Context and forces . . . 69
6.2.3 Solution . . . 70
6.2.4 Structure . . . 70
6.2.5 Consequences . . . 71
6.3 Action Service Factory . . . 73
6.3.1 Intent . . . 73
6.3.2 Context and forces . . . 73
6.3.3 Solution . . . 74
6.3.4 Structure . . . 74
6.3.5 Consequences . . . 74
6.4 Concluding remarks . . . 76

7 Communication **77**
7.1 Overview of communication patterns . . . 77
7.1.1 Remote Procedure Call . . . 77
7.1.2 XML-based RPC . . . 78
7.1.3 Remote Method Invocation . . . 79
7.1.4 Asynchronous Protocols . . . 79
7.2 Channel . . . 80
7.2.1 Intent . . . 80
7.2.2 Context and forces . . . 81
7.2.3 Solution . . . 82
7.2.4 Structure . . . 83
7.2.5 Consequences . . . 85
7.3 Real-time Channel . . . 88
7.3.1 Intent . . . 88
7.3.2 Context and forces . . . 88
7.3.3 Solution . . . 89
7.3.4 Structure . . . 89
7.3.5 Consequences . . . 90
7.4 Streaming Channel . . . 93
7.4.1 Intent . . . 93
7.4.2 Context and forces . . . 93
7.4.3 Solution . . . 94
7.4.4 Structure . . . 95
7.4.5 Consequences . . . 96
7.5 Concluding remarks . . . 97

8 Actors **99**
8.1 Interactive system architecture 99
8.2 Distributed Model-View-Controller 102
8.2.1 Structure 103
8.2.2 MVC variants 105
8.3 Distributed Presentation-Abstraction-Control 106
8.3.1 Structure 107
8.3.2 PAC Variants 108
8.4 Comparing PAC to MVC 109
8.4.1 Coupling of input and output 109
8.4.2 Communication and control 110
8.4.3 Distribution 111
8.5 Actor: a PAC agent 112
8.6 IPML system: an IPML actor 112

III Timing and Mapping **115**

9 Timing **117**
9.1 Background Concepts 118
9.1.1 Natural and synthetic temporal relations 118
9.1.2 Time-dependent and time-independent elements 118
9.1.3 Who has the time? 119
9.1.4 Synchronization reference model 120
9.2 Timing models 122
9.2.1 Instant based models 122
9.2.2 Interval based models 125
9.2.3 Timing models for authoring and directing plays 130
9.3 Action Synchronization Engine 132
9.3.1 Petri nets 132
9.3.2 OCPN 132
9.3.3 ASE 136
9.3.4 Object-oriented implementation of ASE 138
9.4 Get ready just in time 138
9.4.1 Just-in-time approach 139
9.4.2 Action time prediction 140
9.4.3 Dynamic adjustment 140
9.5 Distributed time 141
9.5.1 Clock synchronization 141
9.5.2 Software clocks 141
9.5.3 “Action!” delayed 142
9.6 Concluding remarks 142

10 Mapping 143
10.1 Mapping requirements to resources . . . 143
10.1.1 Preliminaries . . . 144
10.1.2 Specification . . . 146
10.1.3 Presentation resources . . . 148
10.1.4 Mapping . . . 153
10.1.5 Discussion . . . 154
10.2 IPML Mapping . . . 155
10.2.1 Implementing the Simple Switch . . . 155
10.2.2 Dynamic Mapping . . . 156
10.3 concluding remarks . . . 158

IV Evaluation 159

11 Implementations 161
11.1 DeepSea (2001-2003) . . . 161
11.1.1 Content . . . 161
11.1.2 Configuration . . . 164
11.2 TOONS in IPML (2003-2004) . . . 166
11.3 TheInterview (2004) . . . 167
11.4 Mov'in - Emotional Pillows (2005) . . . 169
11.5 Concluding remarks . . . 170

12 Fun and Presence 171
12.1 Introduction . . . 171
12.2 Measurements . . . 172
12.2.1 Fun and Presence . . . 173
12.2.2 Measurement instruments . . . 174
12.3 Experiments and Results . . . 176
12.3.1 Experiment 1 . . . 176
12.3.2 Experiment 2 . . . 181
12.3.3 Results from the interviews (Experiment 1 & 2) . . . 183
12.4 Discussion . . . 185
12.4.1 Level of Control effect . . . 185
12.4.2 Effects of distribution . . . 186
12.4.3 Number of Users effect . . . 187
12.4.4 Content . . . 187
12.4.5 Experimental Design . . . 188
12.5 Concluding remarks . . . 188

13 Culture Matters 189
13.1 Introduction . . . 189
13.1.1 Two cultures studied . . . 190
13.1.2 Distributed interactive media . . . 191
13.1.3 Embodiment and Interaction . . . 192
13.1.4 Research questions . . . 192

13.2 Experiment . . . 192
13.2.1 Measurements . . . 193
13.2.2 Participants . . . 193
13.2.3 Setup . . . 194
13.2.4 Procedure . . . 196
13.3 Results . . . 196
13.4 Discussion . . . 199
13.4.1 Culture effects . . . 199
13.4.2 Embodiment effects . . . 199
13.4.3 Effects of direct touching . . . 200
13.4.4 Future Research . . . 200

14 Conclusions **201**
14.1 Reflection on the design process . . . 201
14.2 Reflection on the resulting design . . . 202

Bibliography **205**

Publications related to this research **227**

Summary **229**

Samenvatting **231**

Curriculum Vitae **233**

The following contents are available on the CD-ROM that accompanies this thesis.

Background materials **235**

Background material A Rapid Robotic Prototyping **237**
A.1 Problem Definition . . . 237
A.1.1 Design challenges . . . 238
A.2 Solution . . . 240
A.2.1 Rapid prototyping in software engineering . . . 240
A.2.2 Difference between robotic prototyping and software prototyping 241
A.2.3 Rapid robotic prototyping techniques . . . 241
A.3 Examples . . . 244
A.3.1 Tony . . . 244
A.3.2 LegoMarine . . . 244
A.3.3 eMuu . . . 245
A.3.4 Mr. Point, Mr. Ghost and Flow Breaker . . . 246
A.4 Discussion . . . 246

Background material B StoryML DTD (Document Type Definition) **247**

Background material C TOONS in StoryML **251**

Background material D DTD of IPML Extension to SMIL **253**
D.1 IPML layout module . . . 253
D.2 IPML Media Object Module . . . 256
D.3 IPML Linking Module . . . 259
D.4 IPML Language Profile DTD driver . . . 261

Background material E Specifications of the Timed Action Pattern **265**
E.1 *ActionService* . . . 265
E.2 *Action* . . . 267
E.3 *ExecutionQueue* . . . 268
E.4 *TimedAction* . . . 270
E.5 *Scheduler* . . . 271
E.6 *ActionServiceProxy* . . . 272
E.7 *TentativeResult* . . . 275

Background material F Specifications of the Synchronizable Object Pattern **277**
F.1 Synchronization states . . . 277
F.2 Coordinates . . . 277
F.3 Repeating positions . . . 278
F.4 Progression directions . . . 279
F.5 Synchronization Events . . . 279
F.6 *Synchronizable* . . . 280
F.7 *ActiveSynchronizable* . . . 288
F.8 *SyncMediaObject* . . . 288
F.9 *ActiveSyncMediaObject* . . . 290
F.10 *Timer* . . . 290
F.11 *TimedSynchronizable* . . . 292
F.12 *TimedMediaObject* . . . 296
F.13 *EventDispatcher* . . . 298
F.14 *EventHandler* . . . 299

Background material G Specifications of the Action Service Factory Pattern **301**
G.1 *SyncService* and concrete actions . . . 301
G.2 *SyncServiceProxy* . . . 302
G.3 Media types . . . 303
G.4 *AbstractSyncFactory* . . . 304
G.5 SyncFactoryImpl . . . 305
G.6 AbstractSyncServiceFactory . . . 306
G.7 SyncServiceFactoryImpl . . . 307
G.8 An Example . . . 307

Background material H Specifications of the Channel Pattern **311**
H.1 *Channel* . . . 311
H.2 *SupplierAdmin* . . . 312
H.3 *ConsumerAdmin* . . . 313
H.4 *PushSupplier* . . . 315
H.5 *ProxyPushSupplier* . . . 315
H.6 *PullSupplier* . . . 316
H.7 *ProxyPullSupplier* . . . 317
H.8 *PushConsumer* . . . 318
H.9 *ProxyPushConsumer* . . . 319
H.10 *PullConsumer* . . . 320
H.11 *ProxyPullConsumer* . . . 320
H.12 An example of the push model . . . 321

Background material I Specifications of the Real-time Channel Pattern **323**
I.1 *QoSProperties* . . . 323
I.2 *Filter* . . . 324
I.3 *Scheduler* . . . 327
I.4 *PriorityQueue* . . . 328
I.5 *Dispatcher* . . . 331
I.6 *RTChannel* . . . 334
I.7 *RTConsumerAdmin* . . . 336
I.8 *RTProxyPushSupplier* . . . 336
I.9 Other components . . . 337

Background material J Specifications of the Streaming Channel Pattern **339**
J.1 *Stream* . . . 339
J.2 *DataSink*, *DataSource* and *DataSourceReceiver* . . . 340
J.3 *StreamingChannel* . . . 341
J.4 *StreamAdmin* . . . 342
J.5 *StreamSupplierAdmin* . . . 343
J.6 *StreamConsumerAdmin* . . . 344
J.7 *StreamSupplier* . . . 345
J.8 *ProxyStreamSupplier* . . . 345
J.9 *StreamConsumer* . . . 346
J.10 *ProxyStreamConsumer* . . . 347

Background material K Specifications of the Distributed MVC Pattern **349**
K.1 *Model* . . . 349
K.2 *ModelDependent* . . . 352
K.3 *View* . . . 354
K.4 *Controller* . . . 354

Background material L Specifications of the Distributed PAC Pattern 357
L.1 *Abstraction* . . . 357
L.2 *Presentation* . . . 359
L.3 *Control* . . . 361
L.4 *PAC* agent . . . 364

Background material M Specifications of the Actor 365
M.1 Refinement of *MediaObject* . . . 365
M.2 *ActorPresentation* . . . 366
M.3 *ActorAbstraction* . . . 367
M.4 *ActorControl* . . . 368

Background material N Firing rules of ASE 371

Background material O Converting IPML to ASE 373
O.1 Action element . . . 373
O.2 Timing containers . . . 374
O.2.1 “seq” . . . 374
O.2.2 “par” . . . 374
O.3 Timing attributes . . . 375
O.3.1 “dur” . . . 375
O.3.2 “begin” . . . 375
O.3.3 “end” . . . 376
O.3.4 “restart” . . . 377
O.3.5 “repeatCount” and “repeatDur” . . . 378
O.4 event based linking . . . 379
O.5 ASE simplification . . . 380

Background material P Correctness of the mapping 383

Glossary 387

Index 400

List of Figures

1.1 Distributed content and distributed interfaces 3
1.2 Spiral design 5

2.1 Materials used in the first meeting with children 12
2.2 TOONS requirements elicitation sessions 13
2.3 Karolina's robot 13
2.4 TOONS Storyboard 15
2.5 Four types of decision points in TOONS 17
2.6 DeepSea Storyboard 21
2.7 DeepSea Setup 22

3.1 Conceptual model of TOONS 26
3.2 Structure of the user interface in digital TV 27
3.3 "Immersive broadcast" components 28
3.4 Simplified model of interactive story 30
3.5 StoryML Object Model 31
3.6 Hierarchy of PAC agents 33
3.7 PAC extended for timed media 34
3.8 Video streaming along the agents 35
3.9 PAC based architecture of the StoryML player 36

4.1 Role types in Beijing opera 41
4.2 19th century drawing of Beijing opera, public domain 42
4.3 SMIL in UML 44
4.4 IPML in UML 47

5.1 Alexander's Patterns 54
5.2 The Observer pattern 57

6.1 Timed Action pattern 68

6.2 Synchronizable Objects 70
6.3 Event based synchronization 71
6.4 Action Service Factory 75

7.1 Client-server communication model 81
7.2 Supplier-Channel-Consumer communication model 82
7.3 Consumer and supplier proxy objects 83
7.4 Data delivery models in Channel pattern 84
7.5 Channel pattern 85
7.6 Channel composition 86
7.7 Real-time Channel 90
7.8 Data flow in a Real-time Channel 91
7.9 Streaming Channel pattern 95
7.10 Simplified view of channels 98

8.1 Seeheim Model 100
8.2 Arch Model 101
8.3 Distributed MVC 104
8.4 MVC variants 105
8.5 PAC in UML 106
8.6 Distributed PAC in a hierarchy 107
8.7 IPML system: an IPML actor 112

9.1 Four-layer reference model 121
9.2 Temporal point net 125
9.3 Temporal Relations 126
9.4 Timeline interpretation of a SMIL fragment 128
9.5 OCPN for an IPML example 134
9.6 Transition controller 137
9.7 An example of the ASE model 138
9.8 Object-oriented implementation of ASE 139

10.1 Formal model structure 146
10.2 A possible configuration 146
10.3 $rate(z, 100, 100K)$ 147
10.4 Structure of the resource model 149
10.5 Four paths and “wire names” 154

11.1 DeepSea Demonstrations 162
11.2 DeepSea content structure 163
11.3 Devices used in the DeepSea prototype 165
11.4 IPML contribution to DeepSea 165
11.5 Tony upgrated 167
11.6 TOONS demonstrations 167
11.7 TheInterview screen shot descriptions 168
11.8 Four selected emotions in Mov’in 169
11.9 Prototype of the emotional pillow actor in Mov’in 170

12.1 DeepSea Evaluation . . . 177
12.2 Mean scores for all measurements in all Control Conditions . . . 178
12.3 Mean scores in the NoneDistribution and LightingDistribution conditions at the level of MediumControl . . . 180
12.4 Mean scores in the NoneDistribution and DisplayDistribution conditions at the level of HighControl . . . 181
12.5 Two users cooperating in DisplayDistribution and HighControl condition . 182
12.6 Mean scores for all measurements in Single and Couple conditions . . 183
12.7 Responses to the question "I like this mode" . . . 184

13.1 Conditions of the experiment with Chinese and Dutch participants . . 193
13.2 Experiment setup . . . 194
13.3 Storyline of TheInterview . . . 195
13.4 A decision point . . . 196
13.5 Means of all measurements in all conditions . . . 198
13.6 Means in the culture and embodiment conditions . . . 198

The following figures are available on the CD-ROM that accompanies this thesis.

A.1 Tony . . . 244
A.2 LegoMarine . . . 245
A.3 eMuu . . . 245
A.4 Game Support Robots . . . 246

F.1 State transitions of a synchronizable object . . . 278
F.2 A subset of *Position* . . . 279

L.1 Data flow in a PAC agent . . . 361
L.2 Inside a PAC agent . . . 364

N.1 Reverse tracking upon a transition enabled by a priority place . . . 372

O.1 Converting "action" to ASE . . . 374
O.2 Converting "seq" to ASE . . . 374
O.3 Converting "par" to ASE . . . 375
O.4 Converting "dur" to ASE . . . 376
O.5 Converting "begin" to ASE . . . 376
O.6 Converting "end" to ASE . . . 377
O.7 Converting "restart" to ASE . . . 378
O.8 Converting "repeatCount" and "repeatDur" to ASE . . . 379
O.9 Converting event based linking to ASE . . . 380
O.10 An example conversion from IPML to ASE . . . 381

List of Tables

4.1 Comparing theater play terms with multimedia 43

9.1 Temporal relations in OCPN 127
9.2 Temporal relations in SMIL 129
9.3 Places in ASE 138

10.1 Middleware standards 158

12.1 Conditions of experiment 1 176
12.2 Mean scores and standard deviations for all measurements 179
12.3 F and p values of ANOVA on the effect of a second participant 183

13.1 Hofstede's (1993) Culture Dimension Scores for Dutch and Chinese . 190
13.2 Mean and standard deviation for all measurements 197
13.3 F and p values for culture and embodiment 199

The following table is available on the CD-ROM that accompanies this thesis.

I.1 QoS properties 324

CHAPTER I

Introduction

This PhD project was proposed in the context of the Philips concept of Ambient Intelligence (AmI). This concept was introduced by Philips Research as a new paradigm in how people interact with technology. It pushes the wishes and promises of ubiquitous computing (Weiser, 1991, 1993) a step further by introducing digital environments that are sensitive, adaptive, and responsive to the presence of people (Aarts, 2004; Aarts and Marzano, 2003; Aarts, Harwig, and Schuurmans, 2001; Harwig and Aarts, 2002). Within such an environment, AmI "will improve the quality of life of people by creating the desired atmosphere and functionality via intelligent, personalized inter-connected systems and services" (Boekhorst, 2002).

In the Media Interaction group at Philip Research, AmI is being approached from different directions through several projects. The HomeLab is a realistic environment as laboratory for electronic in-home systems to explore feasibility and usability with end users. In this environment many projects experiment with the concepts and technologies within the paradigm of AmI. To mention two of them, "PHENOM" created an environment that is aware of the identity, location and intention of people; "Easy Access" developed multi-modal and personalized user interface concepts such as voice control, query by humming, and hand writing recognition (Aarts and Marzano, 2003).

Philips Research is not alone in this domain. Many other companies have similar visions to AmI of Philips. Xerox started "ubiquitous computing" research at Palo Alto Research Center (PARC) in the late 1980s. IBM called it "pervasive computing" in a special issue of *IBM System Journal* in 1999 (Vol. 38, No. 4). IBM has a living laboratory, called "Planet Blue", which focuses on the integration of existing technologies with a wireless infrastructure (IBM, 2005). Carnegie Mellon University's Human Computer Interaction Institute (HCII) is working on "distraction-free ubiquitous computing" in their Project "Aura" (Sousa and Garlan, 2002), "to provide each user with an invisible halo of computing and information services that persists regardless of location"(CMU, 2005). The Massachusetts Institute of Technology (MIT) has a project called "Oxygen", in a vision of the future of "pervasive, human-centered computing", freely available everywhere, like oxygen in the air we breathe (MIT, 2004). HP has a system called “Cooltown” to support “web presence” for people, places and things, addressing the issues of “pervasive”, “ubiquitous”, “nomadic” and “context-aware” computing (Kindberg et al., 2002).

Although these projects and research programs have different focuses and emphases, they are aiming at AmI with the same three key technologies: Ubiquitous Computing, Ubiquitous Communication and Intelligent User Interfaces. Ubiquitous Computing means the integration of microprocessors into everyday objects like furniture, clothing, toys, and even paint. Ubiquitous Communication enables these objects to communicate with each other and the user by means of wireless networking. An Intelligent User Interface enables the inhabitants of the AmI environment to control and interact with the environment in a natural and personalized way.

Inspired by these ongoing research programs, this PhD project assumes that AmI will soon come true and alive in people’s daily lifes instead of being visions and concepts. Given an AmI environment, this project explores the challenge of presenting interactive multimedia to such an environment.

AmI will change the way people use multimedia services. The environment which includes many devices, will play interactive multimedia content to engage people in a more immersive experience than just watching television shows. AmI makes it possible that people interact with not only the environment itself, but also the interactive multimedia via the environment. As Aarts (2004) points out, “the requirements that ambient-intelligent multimedia applications impose on the mechanisms users apply to interact with media call for paradigms substantially different from contemporary interaction concepts”.

For many years, the research and development of multimedia technologies have increasingly focused on models for distributed applications (Buford, 1994; Serpanos and Bouloutas, 2000; Vogel et al., 1995). The term “distributed multimedia” refers to the fact that the content sources of a media presentation are distributed over a network. Now, in the context of AmI, not only are the sources of the multimedia distributed, but the presentation and interaction will also be distributed over devices in the environment. It provides the AmI with an integrated and intelligent user interface, or distributed interfaces (figure 1.1 on the facing page).

“Distributed interfaces” refers to the fact that the multimedia content will be distributed to the networked devices of the environment. These devices should cooperate to give the user the feeling of interacting with one “integrated” interface of the environment, not only the individual devices.

The concept of AmI even frustrates the current definition of multimedia. Everyday devices such as lights, toys and electronic carpets together with traditional audiovisual devices can constitute an immersive environment for a multimedia program. In this environment, people will feel they are involved into the multimedia program instead of sitting in front of a television. The brightness and color of lights, the behavior of toy robots and the changing patterns of electronic carpets will be new types of media.

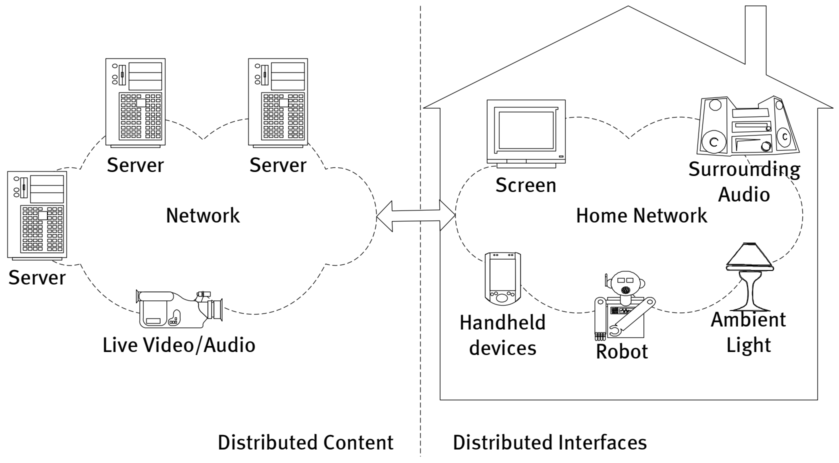


Figure 1.1: Distributed content and distributed interfaces

This PhD project focuses on the structure of multimedia content and the distributed interfaces in the context of AmI, with the following assumptions and limitations:

1. The destination of the media content is an AmI environment, which is closed, yet only open for content delivery and retrieval. An example of such an environment is the home environment where the users consume the media content. This PhD project does not investigate any issues related to the communication and cooperation between two environments.

2. The media content is always available when it is needed, no matter whether it is distributed over a network, or provided by a remote or local server. This means this project does not work on content delivery.

3. The devices involved in the environment are always connected, no matter whether it is wired or wireless connection. This project takes different connection bandwidths into account, but not the underlying network infrastructure.

4. The devices involved carry their own well-designed interfaces. This project does not work on usability issues related to individual devices.

5. The user profiles and preferences are always known and any changes on them will be notified or reported to the system. This project assumes the environment has enough means to collect, save and retrieve the user data.

1.1 Focus of this thesis

With the above assumptions, the hypothesis is formulated in short as follows:

> In an AmI environment, user experience of multimedia can be enriched by structuring both the media content at the production side and the playback system architecture at the user side in a proper way. The structure should enable both the media presentation and the user interaction to be distributed and synchronized over the networked devices in the environment. The presentation and interaction should be adaptive to the profiles and preferences of the users, and the dynamic configurations of the environment.

Therefore, this PhD project therefore focuses on the architectural issues to structure the media content and the system, to enrich the user experiences of multimedia in a distributed environment. Many questions may arise:

1. What constitutes the user's immersive experience of multimedia media?

2. Which basic factors of an AmI environment are important for immersive media experience?

3. By what means will the content authors compose interactive media for many different environments? The authors have to be able to specify the following in their scripts with minimized knowledge of the user environments:
   - Desired environment configurations.
   - Interactive content specification for this environment.

4. Once the content is ready, how will the system playback the interactive media with the cooperation of the user(s) in a way that:
   - makes the best use of the physical environment to match the desired environment on the fly.
   - enables context dependent presentation and interaction. Here the term "context" means the environment configuration, the application context, the user preferences, and other presentation circumstances.
   - synchronizes the media and interaction in the environment according to the script.
   - takes into account the differences among human perception channels such as visual and acoustic perception.

Interpreting these questions as an invitation to design, the main focus is therefore to develop a generic framework to enable presenting interactive media to a networked environment. In particular:

- a media documentation method is to be designed to enable authoring interactive media for a variety of destination environments – the users' homes.

- on top of existing networking technologies and platform architectures, a generic system architecture is to be designed to enable playing the interactive media in a networked environment, with user preferences and dynamic environment configurations taken into account.

1.2 Approach

1.2.1 Spiral model

During the design, a spiral model (Boehm, 1988, see figure 1.2 for a simplified version) was followed. The first spiral was needed to get some first-hand experience and the preliminary requirements. It was concluded with the structure of Interactive Story Markup Language (StoryML) and a demonstrator TOONS, which put the requirements for the second iteration on a stable foundation. More user requirements and technical challenges emerged in the second iteration, grounding the third iteration towards the final design of the Interactive Play Markup Language (IPML) system.

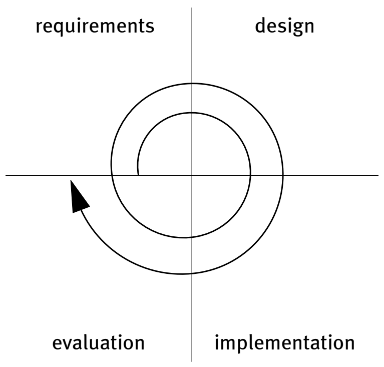


Figure 1.2: Spiral design

Three iterations were completed during this project:

1. A demonstrator called TOONS was developed during the NexTV (2001) project based on the preliminary requirements gathered from the users, which resulted in an experimental software structure (see chapter 3).

2. The design and the development was continued in the ICE-CREAM (2003) project, which resulted in a demonstrator called DeepSea in cooperation with a project partner de Pinxi (2003). It finished with an architecture that is based on the de Pinxi 3D movie engine acting as the central scheduler, and our IPML (see chapter 4 and chapter 11) structure for interfacing devices. Examples were a robotic toy submarine, a portable display, ambient lights and flashing lights. This demonstrator was used for the user evaluation of the concept of interactive movies in distributed environments.

3. The design and development were brought forward after the ICE-CREAM project based on the technical requirements and the experience that were gained from this project. The design was completed with a full implementation of the proposed architecture that was based on open standards and technologies (see chapter 11). Three more demonstrators were built to validate the design, one of which was used to test the effect of the user's culture background on their perception of presence.

More details about the NexTV project and the ICE-CREAM project will be available in chapter 2 when the requirements are presented.

At the beginning of each iteration, after the requirements were clarified, an architecture-based design approach was followed working towards the final technical solutions.

1.2.2 Architecture-based design

An architecture describes the overall technical structure of a system. It consists of software and hardware components in various configurations. An architecture represents the manifestation of the earliest design decisions about a system and is an opportunity for an early validation of the design decisions with respect to qualities (Bass, Clements, and Kazman, 1998).

An architecture based design method is used during this project. The procedures and steps are based on the Architecture Based Design (ABD) method developed by Bachmann et al. (2000). This method is later revised and renamed to Attribute Driven Design (ADD) to emphasize the quality attributes such as performance, modifiability, security, reliability, availability and usability, instead of very specific requirements (Bass, Klein, and Bachmann, 2002; Buchmann and Bass, 2001). This method provides a series of steps for designing conceptual software architecture for complex systems, especially when detailed requirements are not known in advance. The ABD fulfills functional, quality and business requirements at a sufficiently abstract level, based on an understanding of the architectural mechanisms used to achieve the requirements (Bosch, 2000; Buschmann et al., 1996; Gamma, Helm, Johnson, and Vlissides, 1995; Jacobson, Griss, and Jonsson, 1997). For example in multimedia systems, an important class of requirements are the real-time constrains. They are treated as functional requirements since timely presentation of content is an essential function of the system.

The method begins when sufficient requirements are available, and ends when commitments to classes, processes and operating system threads are being made. In general, it provides organization of function, identification of synchronization points for independent threads of control, and allocation of processes to processors.

The main steps of the ABD method are the following (Bachmann et al., 2000):

1. Choose architectural drivers: What combination of the functional, quality and business requirements is most important?

2. Encapsulate functions: translate functional requirements into responsibilities and group them.

3. Determine architecture options: what are the general “styles” or “strategies” to achieve the required qualities?
4. Choose Architectural style: find the collection of component types together with a description of the pattern of interaction, based upon the most important architectural drivers.
5. Allocate Functionality to style: map the responsibility groups onto the collection of component types.
6. Identify/Refine templates: find and examine the common patterns, services or policies in the collection of component types.
7. Verify functionality: Can the use cases and scenarios be achieved?
8. Generate concurrency view: examine activities that may be performed in parallel.
9. Generate deployment view: distribute the system over multiple processors and identify the influences.
10. Verify quality scenarios: Can the quality requirements be achieved?

Detailed description and explanation of all decision to be made in each step can be found in (Bachmann et al., 2000). These steps have to be taken at each level of the decomposition process: from system level to conceptual sub-system level and conceptual component level. Note that it is not the intention that all steps are taken in the above-mentioned order; especially in steps 2 to 4, frequent back-and-forth hopping will occur.

The ABD method was practiced when the pattern-oriented architecture was designed (see part II). Architectural patterns were identified by following the ABD steps, which helped the project to focus on high level structure problems during the design phase without shifting too much to the implementation details.

1.3 Outline of this thesis

This thesis is divided into four parts:

Part I: Requirements and Concepts (chapters 2, 3, 4)

Part II: Architecture Design (chapters 5, 6, 7, 8)

Part III: Timing and Mapping (chapters 9, 10)

Part IV: Evaluation (chapters 11, 12, 13)

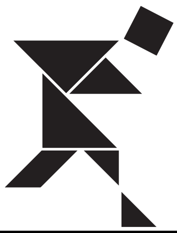
It is worthwhile mentioning that the design problem at hand is not just a technical design for an existing product category. Neither the distributed content nor the distributed home environments are readily available. Yet it is the ambition of the project not only to design the architecture but also to bring it alive through demonstrations, using contents and environments that would be expected in the future. Therefore, the requirements can not be a straightforward list of qualitative and quantitative data. Part I is an exploration into the design of content and environments for a variety of demonstrators.

Part II and part III are much more systematic. They deploy existing research results such as patterns and formal specification methods to arrive at an innovative architecture and finally at a concrete implementation.

The evaluation in part IV goes beyond checking implementation against specification. In view of the innovative nature of distributed media presentations and in view of the fact that AmI is a vision gradually becoming implemented, not yet an established fact, there is hardly any other experience in using such distributed media presentations. Therefore, it is a unique opportunity to use the architecture and the demonstrators to explore several aspects of the user-system interaction. In particular, part IV describes experiments regarding fun and presence in interacting with distributed media content, and the relation between the user's culture background and their feeling of presence.

Part I

Requirements and Concepts



CHAPTER 2

Explorations and Requirements

Chapter 1 has briefly mentioned that this project went through three complete spiral iterations, each starting from a study on user requirements or technical requirements. Since every iteration was an exploration of future possibilities and new technologies, a straightforward and extensive listing of qualitative and quantitative requirements for all types of future systems is not feasible here. Instead, the requirements are summarized as an exploration into the design of the content and environments for the demonstrators.

Section 2.1 summarizes the requirements study and the results in the NexTV project, aiming at building a demonstrator (TOONS) that presents interactive stories in distributed environments. Section 2.2 presents the requirements for the DeepSea application, focusing more on the needs of distributed interfaces and environments. In section 2.3, a number of technical requirements are briefly summarized as the starting point for the design of the final IPML system.

2.1 NexTV and TOONS

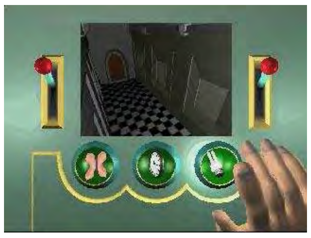
The NexTV (New media consumption in EXtended interactive TeleVision environment, IST-1999-11288) project was funded by the Information Society Technologies programme of the European Commission. NexTV commenced in January 2000 and finished in December 2001. It was consortium of twelve partners from all over the world. Involved in the project were: Philips Research (The Netherlands), The Imperial College of Science, Technology and Medicine (United Kingdom), T-Nova Deutsche Telekom (Germany), FhG FOKUS (Germany), Optibase (Israel), TILAB (Italy), Sony Service Center Europe N.V. (Belgium), KPN Research (The Netherlands), Sun Microsystems (USA), ETRI (Korea), Nederlands Omroepproductie Bedrijf (NOB, The Netherlands) and France Telecom (France).

The NexTV project was to investigate how the new interactive technologies such as MPEG-4, Extensible Markup Language (XML) and Multimedia Home Platform (DVB-MHP) could influence the traditional television broadcasting. It focused on the design and the development of an interactive storytelling application, namely TOONS, for 8-12 years old children. The goal of the application was to enable children to create their own broadcasting programs and their interaction environments by exploration, manipulation and creation of the content elements. A user-centered approach was followed to get the first input from our target user group[1].

2.1.1 Getting the user input

The preliminary user trials were meant to result in a collection of stories created by children and based on their imaginations for how they would like to interact with their story. As a result, an interactive storytelling application was to be built upon the user input. Two meetings were arranged with eight children in total.

In order to elicit as much information as possible, support materials were prepared beforehand, including: a short video sequence with the introduction to the TOONS story created by NOB; a movie compiled out of the assets provided by NOB and additional 2D hand drawings, enabling a limited amount of interaction (figure 2.1(a)); Inspiration cards depicting various objects and representations of emotions (figure 2.1(b)); color prints of the exemplary house interiors (rooms, corridors); drawing paper; crayons, markers; post-it notes.


(a) Test movie

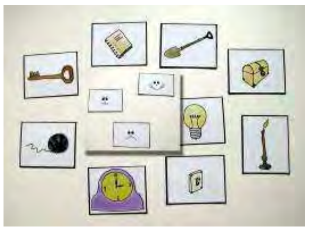

(b) Inspiration cards

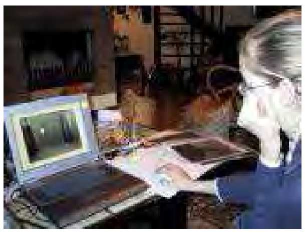
Figure 2.1: Materials used in the first meeting with children

The first meeting was to elicit from children ideas for an interactive story. The children were shown the introduction to the NOB short video sequence and, after a short discussion about the fragment they saw, were asked to write the continuation of the story. They were given one week for this task, until the following Wednesday, when they would gather again and visualize some of the story elements by sketching one of the rooms, painting objects, making story boards, or whatever they choose to do. The children came up with interesting ideas that could be used in the interactive story, such as changing the mood of a room. They created a lot of paintings, drawings and sketches that were later used in the animated movie.

The second meeting was to find out what kind of influence the children would like to have over the interactive story and how. The meeting started with a general introduction to the NexTV project and the interactive story concept. The children

[1]The work presented here was done closely together with Magdalena Bukowska (Bukowska, 2001): we worked out the basic concepts together, she did all the user trials and I provided the working prototypes.

had no problems to understand the idea of being able to influence a TV programme. Next the prepared Macromedia Director movie was shown to them, presenting part of the TOONS story and one possible interaction point (figure 2.2(a)). Afterwards a discussion about possible interaction was conducted in a free talk style. Interesting ideas came up from the conversations, for example highlighted door handles to indicate story paths, apples to be picked up and bitten for story transitions etc.


(a) Interacting with the test movie

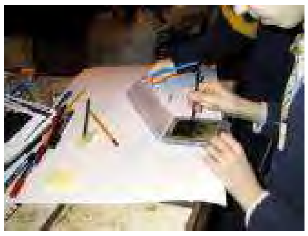

(b) Found a PDA to be interesting

Figure 2.2: TOONS requirements elicitation sessions

The ideas of involving multiple devices for interaction were also brought up by the children, for example the children found a PDA with a touch screen to be an interesting interaction device (figure 2.2(b)) for influencing the story. The children had an idea of using the touch screen to place their names on it, and to customize an on-screen character. Using the touch screen, they would like to construct a character from a set of predefined elements, such as eyes, noses, lips and haircuts. Further, they would like to draw their own character directly on the touch screen, or to draw the character first on a piece of paper and then scan it in using the same device.

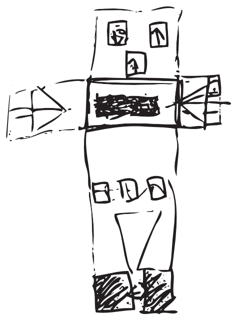


Figure 2.3: Karolina's robot

Another interesting idea was from Karolina, a 12 year old girl. Inspired by the experimenter's question about a physical way of selecting an object on screen, she suggested a robot as the tangible interface device (figure 2.3):

> *"As I understand, there will be a special device sold together with the program that can be used to make choice, right?"* - Karolina

She suggested some buttons on its hands or shoulders as the input channel, and a touch screen in its "belly" as the output channel. The side of the robot would correspond with the side of the screen, i.e. left side of the robot would be used to do something in the left part of the screen, etc. The robot would have a display in its "belly", which could be the touch screen from the PDA, so that she can customize or create her character using the "belly" display, and even scan her drawings by the robot "walking" on her drawings.

They also liked the idea of being able to scan photographs of their faces in and use them as a screen character's face.

> *"Wow, that would be fun!"* – Karolina

2.1.2 Requirements of TOONS

The inputs from the children, both the stories and the interaction concepts, were taken into account for the design and the implementation. A storyboard was created by Bukowska (2001) to communicate the ideas from children to the content creator (NOB) for creating content assets and to the system designers for designing the structures. Figure 2.4 on the next page shows part of the storyboard where the main character in the story needs to make a decision in front of two doors.

Preliminary requirements were identified through the meetings with the children, the elicitation of the content and the concepts, and the creation of the storyboard.

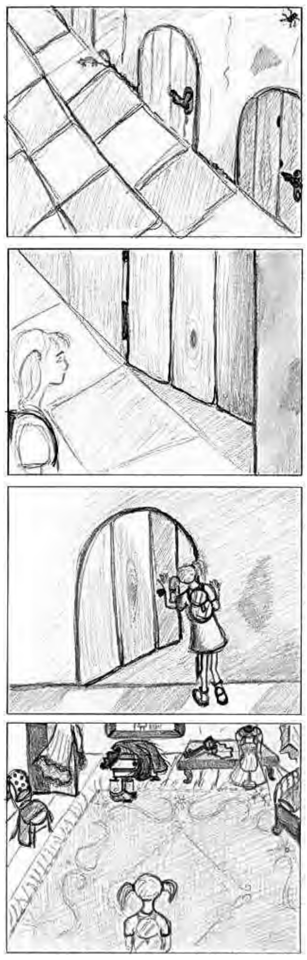
Two types of interaction

Depending on how the interaction is initialized, two types of interaction within the application can be distinguished:

1. *Interaction initialized by the application.* The user can respond to the programme in certain, pre-defined moments. The application invites the user to interact, giving information about an interaction possibility (feed-forward information) and, as soon as the input is given, generates feedback information. The interaction will result in an immediate or delayed change in the programme content, depending on the story scenario. In case of a delayed change, feedback has to be given to the user to confirm the input has been received and understood by the system. This user-system dialogue is often time-constrained. If the user does not provide input within a certain (pre-defined) period, the application will proceed following a default path.
2. *Interaction initialized by the user.* The user has a possibility to make changes throughout the programme, without an explicit invitation from the system side. In this case, the user needs to have prior knowledge about such a possibility. An example of such interaction would be the insertion of the user's own (or a pre-defined) drawing in the story.

Tangible interaction

The children have tangible interaction tools in their possession to interact with the on-screen objects and streams. These interaction devices will provide feed-forward



The girl walks into the house. It is dark. Old wooden doors with rust door handles are closed, except for · · ·

The girl approaches one of the doors which seems to be open.

She cautiously pushes the door open and peeks into the room. As she does it, a strange force draws her inside.

The girl is startled at first by the unusual event but soon forgets about it. She is standing in a colorful room with sets of clothes hanging and lying around as if waiting for someone to put them on. There is an old key lying on the table · · ·

(Storyboard by Magdalena Bukowska)

Figure 2.4: TOONS Storyboard

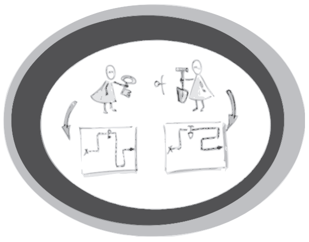
and feedback in order to facilitate the interaction with the story. Feed-forward can be obtained for example through light or sound in the interface device to indicate what can be activated, what actions can be undertaken, and how the actions can be achieved. This information must be provided by the broadcaster along with the story: what actions can be undertaken at what point in the story. To enhance its effect, the story should also supply information needed for the interaction, for example, voice-overs that tell the users that a decision point is at hand. The feedback provided by the interface device should be given immediately after actions are undertaken by the children, for immediate feedback is one of the most important and well-established usability guidelines. The feedback can be in the interface device itself, through sounds ('click' when a button is pressed) or light (an illuminated button that has been activated). In any case, information about the result of the user action should always be immediately presented, so that the users know that their actions did have some effect.

Four types of decision points

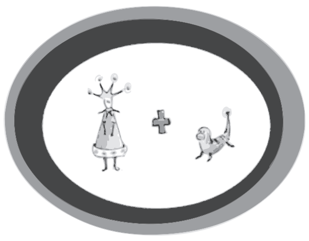
In the TOONS application there are four different types of decision points. These decision points are derived from enabling features of the MPEG-4 object representation. These decision points are [2]:

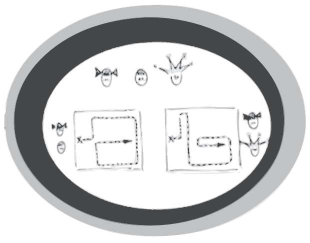
1. *Influencing the storyline by choosing an object.* The story line depends on the object that is chosen by the user, for example a key or a shovel. If the user, for example, chooses the key, the character in the story moves into the secret room through the door with the large lock that the user is now able to open. It is assumed that this type of decision point will support children that are interested in action and adventure in the plot of a story.
2. *Influencing the storyline by choosing an emotional mood or changing a property of an object.* The user can select an emotion for a specific character in the story. This emotion can be a happy or a sad mood. Depending on this mood a different story line will be followed. The moods can be attached to characters but also to objects, rooms and so on. It is assumed that this type of decision point will support children that are interested in the social and emotional developments of characters in a story.
3. *Adding characters or objects to the scene.* The user can add a character to the story. This character will appear in the story, but does not influence the story line. It is assumed that this type of decision point will support children's fantasy and the ability to create one's own story in the context that is provided by the broadcast story.
4. *Influencing the storyline by forming a team or changing the relation between objects.* The user can form a team of two characters from a number of characters. Depending on which characters are in a team, a different story line will be shown. It is assumed that this type of decision point will support children that are interested in the social and emotional development of characters in a story.

[2]The description of these four decision points are edited from Marcelle Stienstra's contributions in the NexTV deliverable *Application Version 1*.


(a) Choosing an object

(b) Choosing an emotion


(c) Adding characters or objects


(d) Forming a team

(Sketches by Marcelle Stienstra)

Figure 2.5: Four types of decision points in TOONS

Technical requirements

The user requirements are collected from the users with limited knowledge about the technical feasibility and possibility. These requirements are about the content (the story), interactivity (decision points) or a specific interface device (Karolina's robot, for example). To develop an application that is based on a generic architecture, these user requirements need to be generalized, with technical feasibility taken into account. Based on the user requirements, the system architecture should support the following:

1. ***Distributed interfaces.*** In the TOONS application, several different components can be distinguished. Full screen audiovisual scenes entertain the users with the story. Interactive objects are present in the scenes, which can listen and react to the user input to personalize their storylines. Graphic user interfaces can be present as overlays on top of the scene, which can be menus, buttons, icons and arbitrary-shaped video clips, or a combination of them.

Different input and output devices can be used to interact with the content. Simple selections and choices can be made with a remote control, while mass data input and complex GUI operations can be done with a remote keyboard and a mouse. A smart card can be used to identify user profiles and to feed the application with predefined configurations. The LED display on the front panel of a set-top box can present extra text information with regard to the real-time streamed content.

The application can also play part of the content and get user responses from some networked devices in the home environment. These devices could be as simple as a bi-directional interface device that can play feed-forward and feedback information that is given by the application, e.g., an interactive toy with touch sensors and sound output. They can also be as sophisticated as robots that have their own behaviors and intelligence. Furthermore, a second screen may be used to present extra media information; for off-line configuration and entertainment, a PC system can be connected.

2. ***Context dependent interaction.*** Here the term "context" means the environment configuration, application context, and the user preferences.

The target system platform can vary from a simple TV set with a set-top box, to a complicated home network environment. The configuration of such an environment is dynamic in both space and time dimensions. The user may activate or introduce new interface devices during the program. The application has to *know* what kind of environment it is running on at every moment and adjust itself on the fly.

The way of interaction may also depend on the application context. For example, in order to illuminate a dark room in the virtual world, a user can simply switch on a real light instead of pressing up or down buttons on a remote control.

However, the user may still choose the remote control because he/she doesn't like to turn the light on, even though there is such a light available. The user, not the system, decides which way of interaction is preferred throughout the program.

3. ***Synchronized media and interaction.*** In an interactive media application, not only the media, but also the interactions are timed and should be synchronized with each other, in an environment which consists of many interface devices. Multiple representations of the content or its parts should be distributed and synchronized on these devices according to their nature and the application semantics. A time dependent change-propagation mechanism is needed for the user-system interaction to ensure that all concerned system components are notified of changes to the content or the configuration, at the right moments in time.

2.1.3 Three versions of TOONS

The TOONS application has three versions of the implementation, based on the same user input and the same content assets created by NOB, each with a different focus:

1. *FhG FOKUS version:* Focus on the interactions initialized by the users and the decision points where the user can influence the storyline choosing an emotion or changing a property of an object. The final implementation allows the children to scan their sketches offline using a scanner, or to take portrait pictures online using a webcam. The scanned images are used to render the objects (for example, as the dressing patterns of the characters), and their own portrait to appear as portraits in the virtual story space.
2. *iPAQ version by Philips:* Focus on the interactions initialized by both the system and the users, and the decision points where the user can influence the story line by choosing an object, and adding characters and objects to the scene using a touch screen Graphical User Interface (GUI) implemented on an iPAQ PDA. The GUI presents the options that the user may choose from, and the objects that the user can add to and remove from the scene.

3. *StoryML version:* Focus on the technical requirements on distribution, context dependency and synchronization. The children can make decisions by interaction with the tangible interface of a toy robot, and the behaviors of the robot are synchronized with the story.

The StoryML version was designed as the first step of this PhD project, towards the design of the architecture for interactive media in distributed environments. The final demonstrator implemented the following story scenario, in an environment which had two presentation devices, a movie player presenting the video content and a toy robot presenting the synchronized behavior and taking the user input:

> *It is 6:00pm in the afternoon. Mark, a 12-year old boy, is watching a storytelling program TOONS showing on a wall in the living room, together with his little toy robot Tony. The lights in the room are changing the brightness following the story. Now in the story, a little girl enters a dark room. The living room becomes dark too. Mark can't see clearly what's happening in that room and he doesn't like the darkness, so he adjusts the light besides him. Both the living room and the room in the story now are illuminated. In the story, the girl is wandering in front of two doors.*
>
> *"Mark, Should I help that lonely girl?" Sleepy Tony is woken up by the lights and seems attracted.*
>
> *"Yes, go ahead." Tony approaches the wall and disappears from the living room. Suddenly he appears in the story. "Hi, Can I help you?" Tony asks the girl.*
>
> *"Yes. I can't decide which door to open."*
>
> *"Left one, Tony!" Mark doesn't know either, but the left door looks nicer. "Mark wants us to go left." Tony opens the left door for the girl.*
>
> *Behind the door there is a beautiful garden with colorful trees and puffy bushes. The sunset beams through the leaves and drops motley shadow into the garden and the living room as well. Mark is surrounded by nice background music. He can hear birds singing their happiness around him.*
>
> *"What a nice garden!" Mark says.*
>
> ...

2.2 ICE-CREAM and DeepSea

ICE-CREAM stands for "Interactive Consumption of Entertainment in Consumer Responsive, Engaging & Active Media". The ICE-CREAM project was about designing compelling experiences for end-users based on enabling technologies for interactive media and by extending the notion of interaction, exploiting domestic activities and familiar settings, and by making the user environment part of the visual experience.

The ICE-CREAM project was set up as a follow-up of NexTV. It was also funded by the Information Society Technologies programme of the European Commission (IST-2000-28298). ICE-CREAM started in January 2001 and finished in December 2003. It was a consortium of twelve partners: Philips Research (The Netherlands), de Pinxi (Belgium), Nederlands Omroepproductie Bedrijf (NOB, The Netherlands), The Imperial College of Science, Technology and Medicine (United Kingdom), Philips Research France, Tomorrow Focus AG (Germany), FhG FOKUS (Germany), Bitmanagement Software GmbH (Germany), Symah Vision (France), and EusKaltel (Spain).

One of the goals of ICE-CREAM project was to continue the exploration of the new technologies in the area of interactive media in distributed AmI environments. Rather than starting a new design cycle from scratch, the exploration took not only the requirements gathered from the NexTV project ahead, but also the experience and lessons learnt from the NexTV project as input, especially those from the design and development of the TOONS application. The following aspects were considered important for the next design cycle:

1. More different types of distributed interface devices should be included in the environment for interactive media. In TOONS, although different interface devices were introduced, they were used in separate demonstrator implementations. Integration of these distributed interfaces needed to be done.
2. TOONS focused on a storytelling application, and the designs of the demonstrators and their architectures was influenced by the structure of this particular type of content. The new design should broaden its application area.
3. A linear narrative structure was used in all TOONS implementations. The new design should also incorporate non-linear structures.
4. The interactive media application should target not only individual users as in TOONS, but also more the multiple users cooperating in a family setting.
5. User evaluation needed to be done based on an integrated implementation to investigate the effects of the distribution on the end user's experience.

2.2.1 DeepSea scenario

The carrier of this cycle of exploration was the DeepSea application. DeepSea is a prototype of an interactive multimedia application. Its scenario characterizes the future of interactive content broadcasting: "it combines video streams, computer graphics, texts, still pictures, sound tracks and multilingual options; it proposes a social approach to content consumption, a new intuitive user interface deployment, and opens the doors to new narrative business models." (ICE-CREAM, 2003)

This application is about enhancing a video broadcast with fictional content, where 3D-graphics and animation are used to realize the fictional effects. The focus of this application is on the application structure and the complexity of the interaction. In this application the content is presented in a non-linear fashion, in such a way that multiple users can watch different facets of the story depending on their different interests. The advantages of such an approach are that content can be re-used within the same program and that users with different interests (for example, a group of friends, the members of a family) can watch the program together at the same time, while each gets their personal flavor. Such programs can provide adventure, education and entertainment at the same time. The content of the story is about deep-sea nature.

A storyboard based scenario was developed by de Pinxi at the beginning of the design of the DeepSea demonstrator as the content requirement (see figure 2.6).

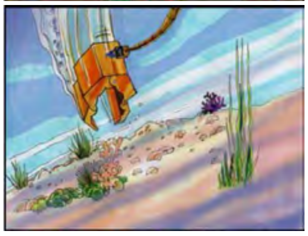
The program starts with the approach to the archaeological site: the shipwreck of the famous Batavia. The camera follows the track made of buoys.

During the navigation, the audience can access additional information on the objects: animals, ship, shells etc. They select them, and get relevant visual (video, stills, texts) or audio information.

The audience can switch to the game mode, and control the navigation of the submarine. They have to follow the color indication of the buoys (red for left, green for right), and avoid the submarine mines.

The audience can switch to the discovery mode, with a view of the inside of the sub (one actor). They discover the site. They launch their robot to go inside the wreck; they follow the robot.

The audience can switch to the game mode, they have to use the robot to pick up an object, and they have to avoid the enemies and obstacles.



The submarine collects the treasure and is brought to the surface by the crane.

Figure 2.6: DeepSea Storyboard

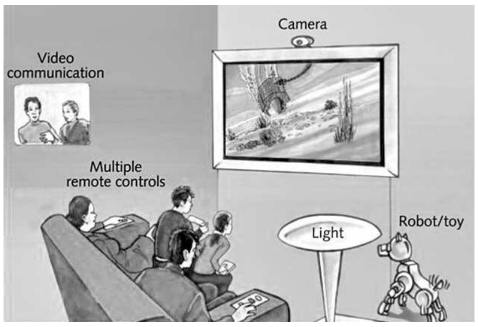


Figure 2.7: DeepSea Setup

2.2.2 Functional requirements

Structure

Using various user interfaces, the audience should be able to interact with the program triggering different facets of the story, hot spots or special effects. The fiction should not be a linear video stream, but should be designed to give the audience a choice between three ways of enjoying the story, depending on their moods or desires:

- The *automated mode* is linear and resembles "classical" video fiction, but providing all the enhancements of the home theater to the story (toy robots, multiple displays, ambience lighting . . .)
- The *discovery mode* presents cultural or scientific information about the fictional world;
- The *game mode* allows the audience to play an immersive adventure game.

Users

The prototype should provide as much as possible the functionality that a full scale of MPEG-4 setup should contain. The application should be designed for small groups like a family unit or a group of friends.

Distribution

The setup should not only encompass the "technical devices" or "the TV" but all the ambience and accessories of a cosy and immersive room: a *home theater*. This should be a living room or a home theater with all comfort: sofas, ambient lights, big screen and special user interface devices to access the functions (figure 2.7).

2.2.3 Levels of interaction

There are different levels of interactivity in this scenario that should be implemented:

1. *Composition of viewing experience (navigating)*. The users should be able to switch between different parts of the presented material, therefore depending on the choices they will receive their own personalized flow of the content.

2. *Influencing the scene composition (interacting with objects).* During watching the main documentary program the user should be able to access additional information about the presented objects. Depending on the devices in use as well as on the user's choice, additional information related to the presented objects would be displayed on the TV screen or on the secondary display, for example, the screen on the bi-directional remote control.
3. *Cooperative interaction.* In the game mode the users should be able to interact with the content, navigating the submarine or the robot in the underwater world. Multiple users should be able to navigate the vehicle at the same time using their own interaction devices. They have to cooperate in order to achieve their common goal.

2.3 Requirements after DeepSea

The design and the development of the DeepSea application were a cooperative effort between Philips and de Pinxi, and successively resulted in a working demonstrator that implemented the concepts of distributed and interactive content in a home theater setting. De Pinxi as an interactive 3D movie provider not only contributed to the project with the content and a 3D movie engine to render the content, but also brought in their expertise in interactive and immersive theaters to create impressive lighting and sound effects. This PhD project, as part of the contribution from the Philips side, focused on the architectural design and also took part in the implementation to carry out the design concepts in practice.

Most of the architectural design concepts presented later in part II, that is, the structure of actors, actions and communication channels, were brought into practice by implementing several distributed interface devices (actors) to render (to perform) the content elements (actions). These devices included a lighting controller for ambient lighting effects, a software component on Philips iPronto for graphical user interfaces, and a robotic submarine for robotic behaviors and navigation control (see section 11.1 for details). The communication channels were used to send synchronization commands to and detect the user interaction from these actors.

Due to limited time, the timing and mapping concepts (presented later in part III) was not implemented for DeepSea. De Pinxi did a great job to embed the scheduling tasks in their proprietary 3D movie engine for these distributed interface devices. Within the time frame of the ICE-CREAM project user evaluations were carried out to investigate the effects of distributed interactive media on the user's experience (see chapter 12 on page 171 for the details about the user evaluation).

After the ICE-CREAM project, it was decided to undertake another design cycle to complete the design with a scripting language that can be used to compose an interactive presentation without knowing the configuration of an environment, with a timing engine that synchronizes the presentation and the user interaction, and with a mapping engine that assigns presentation tasks to the devices that are actually available in the environment. Although the design of these components had already started during the ICE-CREAM project, it had not been completed the design and put into practice at the end of the ICE-CREAM project.

There was another motivation to continue the project with a complete design. In the DeepSea evaluation, variations in user's fun and presence experience were observed, and it was suspected that the characteristics of the users themselves would have an effect on the experience in a distributed environment. For example, there was a long time believe that the user's culture background would have an effect on the user's feeling of presence. However no empirical study had been done to confirm this conjecture, especially in distributed AmI environments. To investigate this interesting topic, it was necessary to complete design and implementation so that an interactive movie could be made and the factors for the experiment could be manipulated.

To summarize, the requirements for the third design cycle were:

1. Complete the design of the scripting language, the timing and mapping engines.
2. Implement at least one demonstrator to visualize the design.
3. Use the demonstrator to investigate the effect of the user's culture background on the feeling of presence, in a distributed setting.

2.4 Rapid Robotic Prototyping

Throughout the project, several types of interface devices for distributed presentation and interaction were designed and implemented. In chapter 1, a generic design question was formulated, which is parameterized over the nature of the devices that will deliver the content. But how can such an abstraction of all possible future devices can be dealt with? This is barely usable in discussions or experiments with users. Of course one can work with displays, like present-day home theaters with handheld devices such as remote controls and PDAs. But in future there will be more. Here a working hypothesis is taken, that robots will be a feasible option to appear among the home devices, and if future users prefer not to have robots at home, from a technical and behavioral viewpoint, many future products certainly do share important characteristics with robots: embodiment, autonomy, and interactivity. Therefore background material A contains an exposé about rapid robotic prototyping, a method that was used during the project to define the requirements of robotic interaction devices.

Rapid prototyping is a powerful method for defining the user requirements for interactive robots, as it is in software engineering. Many prototyping techniques from software engineering are still valid, but need to be adjusted for the nature of robots and the tactile human-robot interaction. In our experience, using robotic kits simplifies and accelerates the prototyping process.

2.5 Concluding remarks

In this chapter the user requirements and the technical requirements were explored for every design cycle. Next the first attempt of the distributed architecture design for interactive media is presented: the StoryML implementation of TOONS, which is based on the requirements described in section 2.1.2 of this chapter.

CHAPTER 3

StoryML[1]

Based on the requirements presented in section 2.1.2, a conceptual design of the interactive story was first developed as the input for system design, and existing technologies and architectures were also evaluated to see if there was a solution that can be taken from the shelf. The design and implementation of the system followed the Object-oriented methodology. The system was implemented using a specific set of technologies, i.e., XML and Java based technologies.

StoryML refers to not only the scripting language designed during the NexTV project for creating interactive stories, but also the player for presenting the created story. It was the first attempt towards a design for a distributed architecture for immersive media. The resulted architecture design and demonstrator had provided with more experiences and insights on the issue than a generic and flexible solution. Nevertheless, the design and implementation StoryML has contributed significantly to the final IPML design, by both providing these experiences and insights, and generating more detailed technical requirements.

3.1 Conceptual model of TOONS

Figure 3.1 on the following page shows the conceptual model of TOONS derived from the requirements collected from the user study. This model consists of storylines and dialogs. The storylines comprise the non-interactive parts in the video stream. The dialogs comprise the parts in which the user can interact with objects in the stream to make decisions. A dialog consists of a feed-forward and a feedback part and a decision point. The story starts with an introduction or opening sequence, followed by a set-up sequence in which the user can customize the objects, by for example choosing different appearances for the main character. In the middle of the sequence there are several decision points where the user can choose from different options in the story. For example, to open one of the two doors presented in the scene, the user can knock

[1]This chapter is based on a paper published in the proceeding of the 21st IASTED International Conference on Applied Informatics (Hu and Feijs, 2003a, AI 2003).

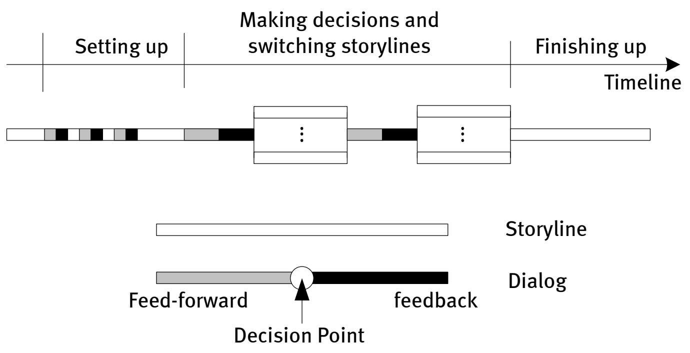


Figure 3.1: Conceptual model of TOONS

on corresponding buttons on a console. Different decisions at the decision points will lead to different storylines. The story ends with a finishing sequence that promises more episodes to come or that the whole story has just finished.

This conceptual model needs to be implemented in distributed settings, with multiple devices as presentation terminals and controlling interfaces (see section 2.1). To meet these requirements, the first question to be answered is how to describe such an interactive story so that it can be interacted with and played back in a distributed environment, and the second question is how to play back the story in a distributed environment with dynamic configurations. To answer the first question, this project first looked at existing open standards, notably Synchronized Multimedia Integration Language (SMIL) and MPEG-4, to see whether there had been a solution for distributed interactive storytelling. For the second question, this project also started from exploring existing architectures, especially in the domain of interactive television (TOONS was aimed at an application in this domain). Next the findings are summarized.

3.2 When technologies meet the requirements

3.2.1 Open standards for synchronized multimedia

SMIL and MPEG-4 are contemporary technologies in the area of synchronized multimedia (Battista, Casalino, and Lande, 1999, 2000). SMIL focuses on Internet applications and enables simple authoring of interactive audiovisual presentations, whereas MPEG-4 is a superset of technologies building on the proven success in digital television, interactive graphics applications and also interactive multimedia for the Web. Both were the most versatile open standards available at the time this project looked for a technology to compose the distributed interactive story.

But both were challenged by the requirement for distributed interactions. It requires that the technology is first of all able to describe the distribution of the interaction and the media objects over multiple devices in an integrated environment. The Binary Format for Scenes (BIFS) in MPEG-4 emphasizes the composition of

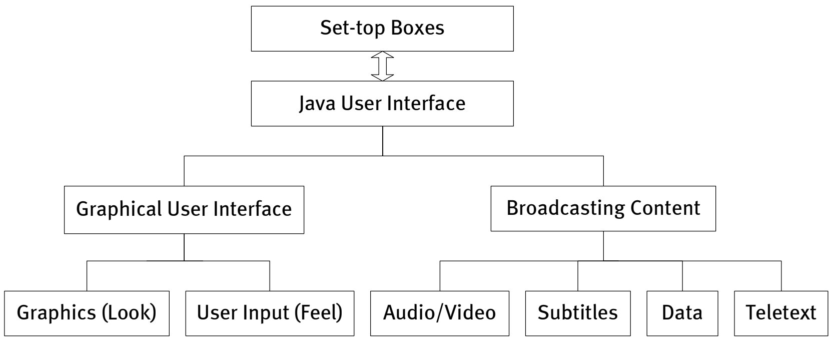


Figure 3.2: Structure of the user interface in digital TV

media objects on one rendering device. It doesn't take multiple devices into account, nor does it have a notation for distributed interfaces.

SMIL 2.0 introduces the MultiWindowLayout module, which contains elements and attributes providing for creation and control of multiple top level windows (Rutledge, 2001). This is very promising and comes closer to the requirements of distributed content interaction. Although these top level windows are supposed to be on the same rendering device, they can to some extent, be recognized as software interface components which have the same capability.

TOONS intends to make use of multiple interface devices with different capabilities, i.e., an audiovisual device and a tangible interface device. In this sense, neither MPEG-4 nor SMIL can fully meet the requirements if directly taken from the shelf.

3.2.2 Architectures for interactive television

Since TOONS was intended to be an application for interactive digital television, this project also looked at up-to-date digital TV interfaces and system architectures.

Future TV

In Future TV (1999), a typical user interface structure for digital TV applications is introduced. It includes not only graphics but also timed media. Figure 3.2 illustrates this basic user interface structure for interactive television services or applications on a set-top box (Eronen and Vuorimaa, 2000; Koenen, 1999; Vuorimaa, 2000). The user interface is composed of a Graphical User Interface (GUI) and broadcasting content. The GUI includes graphics and user input as the so called Look & Feel.

In this structure, however, the GUI is not part of the content. The user may select different content by manipulating GUI widgets. This interaction does not enable direct manipulation of anything inside the content. What the interface should look like and how the user can operate it depend on the implementation of the local platform.

This structure has no possibilities whatsoever for content presentation on distributed interfaces, nor for synchronization between the media and the interaction.

Immersive broadcast

Immersive broadcast is a generic term for interactive multimedia applications mainly targeting enhanced digital television programs, introduced in a white paper from Philips Research (Mallart, 1999). An immersive broadcast application for sports events is presented by Herrmann (2000). In this application, the consumer can compose his own personal program from a variety of streams of audiovisual and graphics data. Conceptually, video clips, text and graphics are overlaid on top of the TV program to provide a richer and more compelling experience for the viewer.

The components of an immersive broadcast application can be categorized into content presentation components (views, highlights and the overlay) and content navigation components (figure 3.3). Compared to the structure shown in figure 3.2 on the previous page, here the user interface components are a part of the broadcasting content. The user interaction will influence the presentation of the live or stored content, or content related information.

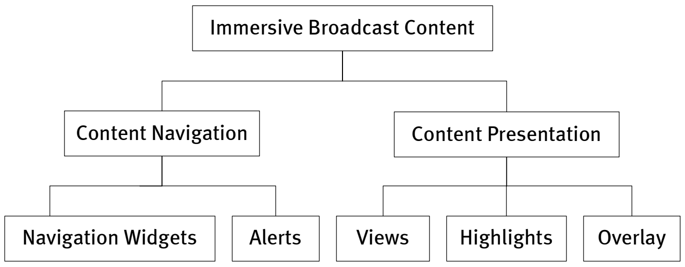


Figure 3.3: “Immersive broadcast” components

This structure does not take into account the distributed presentation and interaction. The user interaction remains at the level of content navigation, which can not change directly what is in the content.

In short, the current technologies described above can hardly satisfy the requirements for the distributed media. A different approach was needed.

3.3 Design of StoryML

As already mentioned, the SMIL approach comes close to the requirements. Its concept of separating the layout concerns of the media objects from their timing and synchronization behavior fits very well to the needs of distributing media objects to multiple devices and synchronizing the presentations across these devices. However SMIL as it is does not allow multiple presentation devices to be included in one integrated presentation. Further, it is designed for presentations that have a predefined layout. This is too concrete for the dynamic layout settings, in this case, for the dynamic environment configurations. Hence it was decided to design a new scripting language based on XML, with a more abstract representation for storytelling applications.

3.3.1 Why XML

Based on the object-oriented story model, an XML based specification language, i.e., StoryML, is developed for authoring, serving, delivering and presenting an interactive story. The common markup languages currently in use are Standard Generalized Markup Language (SGML) and HyperText Markup Language (HTML). SGML is a standard system for defining and using document formats (ISO, 1986) and HTML is a language used for hypertext linking, multimedia and displaying of simple documents on the Web (Raggett, Le Hors, and Jacobs, 1999).

Extensible Markup Language (XML) is designed to provide an easy-to-write, easy-to-interpret, and easy-to-implement subset of SGML (Yergeau et al., 2004). It is not a fixed format like HTML. XML is a meta-language used to define other markup languages for structured documents. Structured documents are those that contain content stored hierarchically, in a specified format. In this sense, HTML is just one of the SGML or XML applications. XML is designed so that a particular markup language, such as StoryML, meets the application needs more quickly, efficiently and logically.

To summarize, XML is used because of the following features:

Extensible: XML provides the ability to extend a language with custom markup tags.

Data and information exchange: XML provides a common language for people and systems to exchange data, because it's human-readable and independent of programming languages and computer platforms.

Describes data: XML describes data with markup, making it easier to process.

Separation of content and format: Authors of XML don't need to be concerned with formatting. Yet XML can be presented in different formats with stylesheets.

Easy to read and process: XML is plain text and is very structured and hierarchical, making it easy to access and process.

3.3.2 Simplified model of interactive story

Figure 3.4 shows the simplified interactive story model which is directly derived from the conceptual model (figure 3.1 on page 26), based on the idea of simplifying the user interaction as switching between storylines. The switching concept was a design decision made by all NexTV partners at the early stage of the project.

An interactive story consists of multiple *storylines* and the user can influence these storylines by switching among them. Multiple linear dialogs compose the interaction between the user and the story. A dialog always starts with the system giving feed-forward information. The user makes decisions or choices and then the system shows immediate feedback information to the user. The dialog results in switching among the storylines, or changes in one or more storylines. Comparing to the conceptual model, this model simplifies the structure of the interactive story.

This model explicitly defines the storyline as a primitive story component which has the same temporal dimension as the story has, that is, starting or stopping a story means starting or stopping these storylines altogether at the same time.

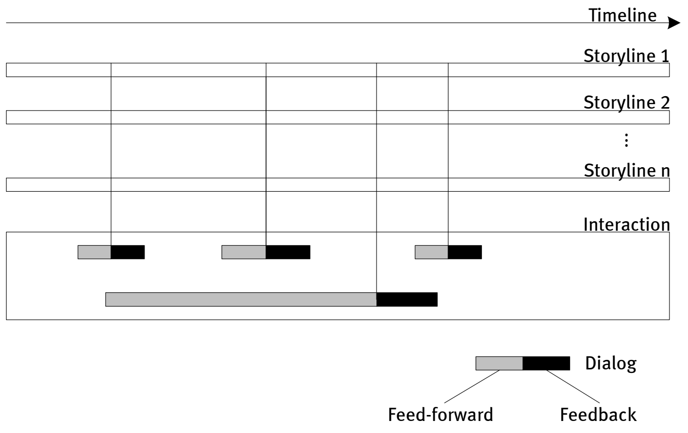


Figure 3.4: Simplified model of interactive story

The *feed-forward* and *feedback* components are separated from the storyline as different types of primitive components because they have different temporal behavior. Different from the storylines, when to start a feedback component or when to stop a feedback component depend on when the user will respond during a dialog.

3.3.3 Environment and actors

The interactive story will be played in an environment which consists of multiple networked devices, such as audiovisual screens, surrounding audio systems, ambient lights, and robotic toys. These devices are abstracted as *actors*[2].

An actor is a self-contained entity which has capabilities of data processing and user interaction. Its input and output facilities form an interface that a user can interact with. An actor is able to abstract the user inputs as events and to communicate with other actors. An actor can be present in an environment as a software entity, alive in a computer system or embodied in a hardware device.

An *environment* is where these actors reside. It defines the preferred configuration of actors. However the physical environment may change during the runtime. The StoryML must map the prescribed environment to the physical one, and send tasks assigned to the actors to the devices in the physical environment. These tasks are, for example, rendering media objects, or reporting the user responses during the dialogs.

3.3.4 Media Objects

Storylines, feed-forward and feedback components are all *media objects*. A media object is defined as a data stream which can be rendered by the actors in the environment, and can be perceived by the user via any or multiple channels of perception.

[2]They were called *interactors* in earlier versions of StoryML.

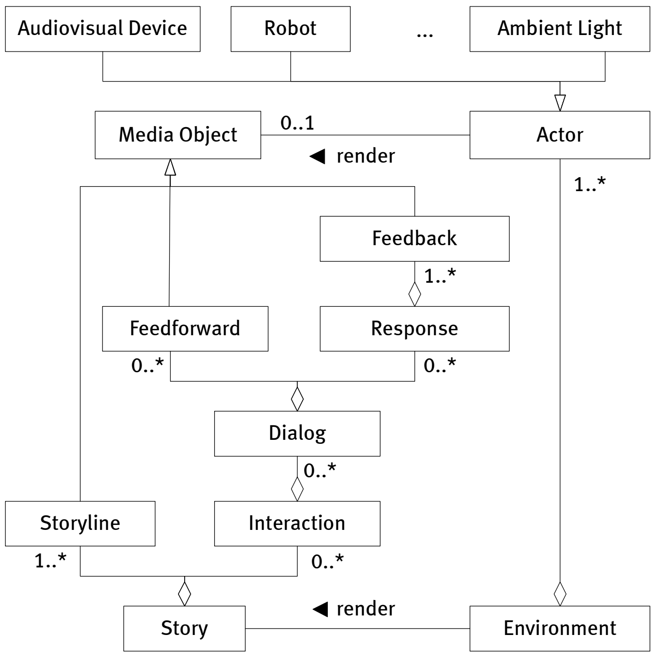


Figure 3.5: StoryML Object Model

Traditional media objects are audiovisual objects, such as audio, video, text, and 2D/3D graphic objects. Some new standards for example MPEG-4 have introduced new media objects that have a higher level of abstraction, e.g., face and body animations (Eronen and Vuorimaa, 2000; Koenen, 1999; Vuorimaa, 2000). Here goes one step further: in StoryML, a media object can be even more abstract. Facial expressions, robotic behaviors, and even emotional modes, can be defined as a media object as long as they can be interpreted and rendered by an actor.

The abstraction of media objects provides with possibilities for content producers to describe a story at a high level without knowing the details of the environment, e.g., the content producers can specify a robot to show a 'happy' behavior without the need to know how the actual robot will render the 'happiness'. It solely depends on the configuration of the environment and the implementation of the robot.

Figure 3.5 shows the object-oriented model of StoryML. It reflects many concepts directly from the conceptual model (see figure 3.1) and its simplified version (see figure 3.4 on the facing page). The major objective of doing so is to make StoryML an easy authoring language for content producers. The Document Type Definition (DTD) of StoryML documents can be found in background material B.

An example is given in background material C. It describes a TV show to be presented in an environment with two actors involved, i.e., an audiovisual screen and the toy robot Tony. Tony will be 'woken up' shortly after the show has started. The user will be invited to help the main character in the show to make a decision (for example "left" or "right") at a certain period of time by playing with Tony.

3.3.5 Timeline

In StoryML, media objects and interaction dialogs refer to an implicit timeline by specifying their starting and stopping point in time. The metaphor behind it can be easily understood by comparing with the conceptual model of the interactive story. Synchronizing objects by means of a timeline can be easily understood by the authors because it reflects very well our one dimensional perception of time. Defining the beginning of a video presentation to an audiovisual actor in a story requires no knowledge of the related video frames. The timeline approach is therefore intuitive and easy to use in authoring situations.

3.4 Implementation of StoryML Player

StoryML has been defined as a solution for writing interactive stories. Now the task is to design an appropriate software architecture for the StoryML player. A Presentation-Abstraction-Control (PAC) based architecture (Coutaz, 1997) is chosen.

3.4.1 PAC based architecture

Many interactive architectures have been developed along the lines of the object-oriented and the event driven paradigms. Model-View-Controller (MVC) and PAC are the most popular and often used ones. Later chapter 8 will provide a detailed comparison. Here let's take a brief look at why PAC is chosen.

The MVC model divides an interactive agent into three components: model, view and controller, which respectively denotes processing, output and input. The model component encapsulates core data and functionality. View components display information to the user. A View obtains the data from the model. There can be multiple views, each of which has an associated controller component. Controllers receive input, usually as events that encode hardware signals from input devices.

In the PAC architecture, an agent has a presentation component for its perceivable input and output behavior, an abstraction component for its function core, and a control to build dependencies among PAC agents. The control of an agent is in charge of the communication with other agents and between the abstract and the presentation in the agent itself. In PAC, the abstraction and presentation components of the agents are not authorized to directly communicate with each other or with other agents. An interactive application is modeled as a set of PAC agents whose communication scheme forms a hierarchy (see figure 3.6 on the next page).

The PAC based architecture is considered more suitable for the StoryML player than MVC, because of the following reasons:

1. StoryML involves independent devices as actors. It should be able to adapt to the changing configuration. PAC can meet this requirement by separating self-reliant subtasks of a system into cooperating but loosely-coupled agents. Individual PAC agents provide their own human-computer interaction. This allows the development of a dedicated data model and user interface for each semantically independent components within the system. PAC agents can be distributed easily to different threads, processes or machines.

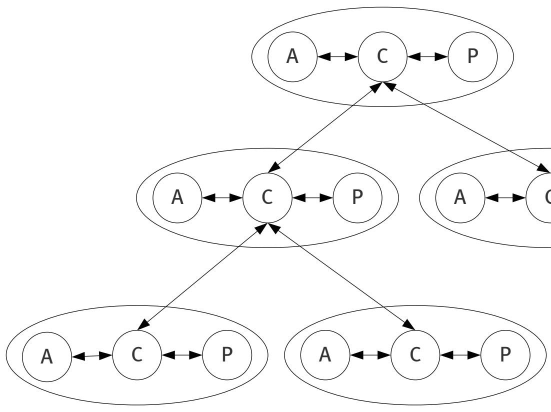


Figure 3.6: Hierarchy of PAC agents

2. The PAC based architecture emphasizes the communication and cooperation among agents and inside an agent with a mediating control component. It is crucial to have such a mechanism for a distributed application like the StoryML player. All agents communicate with each other via their control component with a pre-defined interface such that existing agents can dynamically register new PAC agents to the system to ensure communication and cooperation.
3. The input and output channels of the individual actors are often coupled. In MVC, controller and view are separate but closely-related components. In the PAC architecture this intimate relationship between the input and output is taken into account. It considers the user accessible part of the system as one presentation component.
4. The StoryML player has to facilitate content based interaction, which means that the user can interact with interactive media objects in the content. Possible user controls are often embedded in the media objects and together they form an entity, which will be rendered by one of the actors. At a conceptual level, this request can be easily assigned to the presentation component. Separating the attached operation from the media object would increase the complexity.

3.4.2 Extending PAC for timed media

However the overhead in the communication between PAC agents may impact the efficiency. For example, if a bottom-level agent retrieves data from the top-level agent, all the intermediate-level agents along the path from the bottom to the top of the PAC hierarchy are involved in this data transportation. If agents are distributed, data transfer requires inter-process communication, together with marshaling, un-marshaling, fragmentation and re-assembling of data (Buschmann et al., 1996).

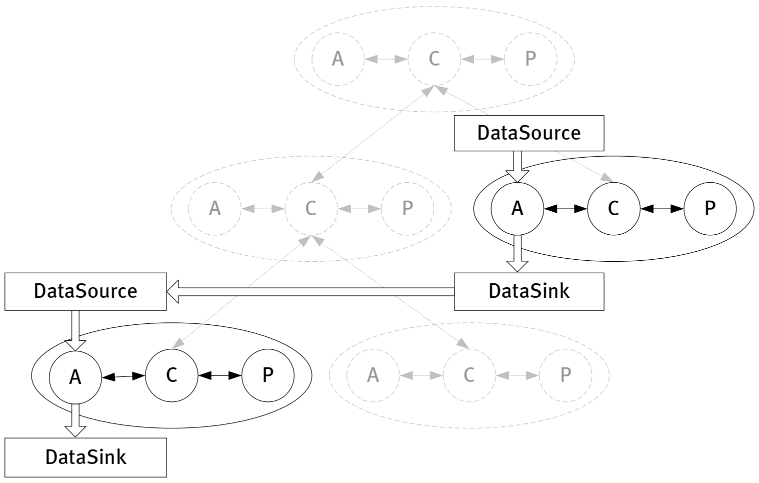


Figure 3.7: PAC extended for timed media

To overcome this potential pitfall, the StoryML player extends the abstraction component. For timed media, each abstraction component is considered as a media processor, which takes a DataSource as input, processing the media data, and then outputs the processed media data to a presentation component or to its DateSink (figure 3.7). DataSink and DateSource are connected through streaming channels.

Regarding the PAC hierarchy as a network, an agent with a DataSink can be viewed as a streaming media server and those with a DateSource can be viewed as streaming media clients. A direct streaming channel can be built between a DataSink and a DateSource and the media data can be streamed through the channel with real-time streaming protocols. The streaming channel between the abstraction and the presentation components can also be built inside a PAC agent but only when the agent needs to present the media, and in this case, the presentation component directly renders the media onto the physical interface without much media processing. Streaming channels can be built and cut off only by the control components. Thus, the control hierarchy remains intact. A detailed description about how this can be done is presented later in section 7.4 of chapter 7.

The idea is to separate the overloaded media streaming tasks from the PAC control hierarchy. In figure 3.9 on page 36, streaming channels are built among the content prefetcher, the virtual audiovisual actor and the physical audiovisual device to stream the massive data and allocate the data processing, under the supervision of the hierarchy. As an example shown in figure 3.8 on the facing page, the content pre-fetcher caches enough raw MPEG-4 data to ensure the presentation can be immediately started at a certain point in time, and keeps the data fetching a smooth and stable streamline. The prefetcher establishes a DateSource for each Media Object. When the audiovisual actor needs to present the content, the upper layer agents connect the related DataSink with a DateSource of the virtual audiovisual actor, thus a streaming channel is built and the streaming task is later carried out among only

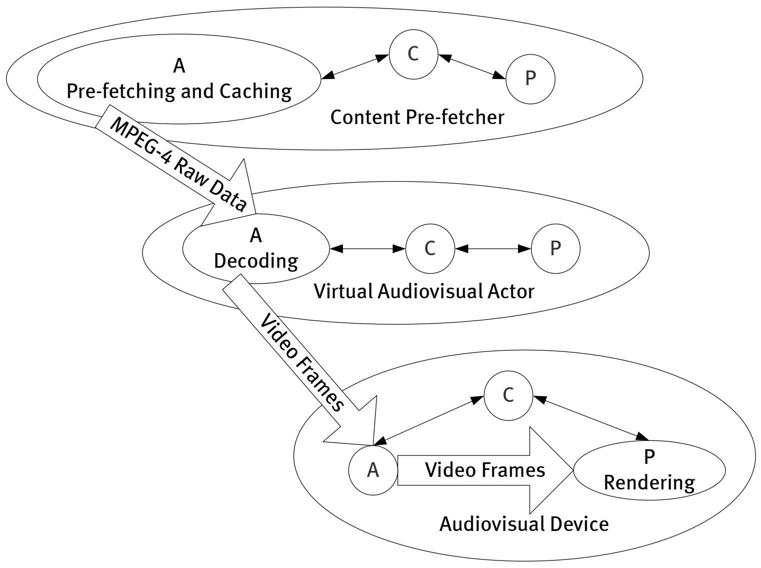


Figure 3.8: Video streaming along the agents

the related agents. The virtual audiovisual actor undertakes the MPEG-4 decoding and then passes the decoded data to the real actor. The real actor connects the video frames to its presentation component, and finally presents the media to the user.

3.4.3 Architecture of the StoryML Player

Figure 3.9 on the next page shows the hierarchical structure of the StoryML player. The content portal sets up the connection to content servers and provides the system with the content. The content prefetcher overcomes the start latency by prefetching a certain amount of data and ensures that the media objects are ready to start at specified moments.

An XML parser first parses the StoryML document into Document Object Model (DOM) objects and then the StoryML parser translates the DOM objects into internal representations. The StoryML player also maintains a timeline controller, which plays an important role in synchronization.

The bottom level agents indicate different physical interface devices. These physical devices are often equipped with embedded processors, memory, and possibly with some input and output accessories. An arbitrary number of physical agents can be added to the architecture at this level.

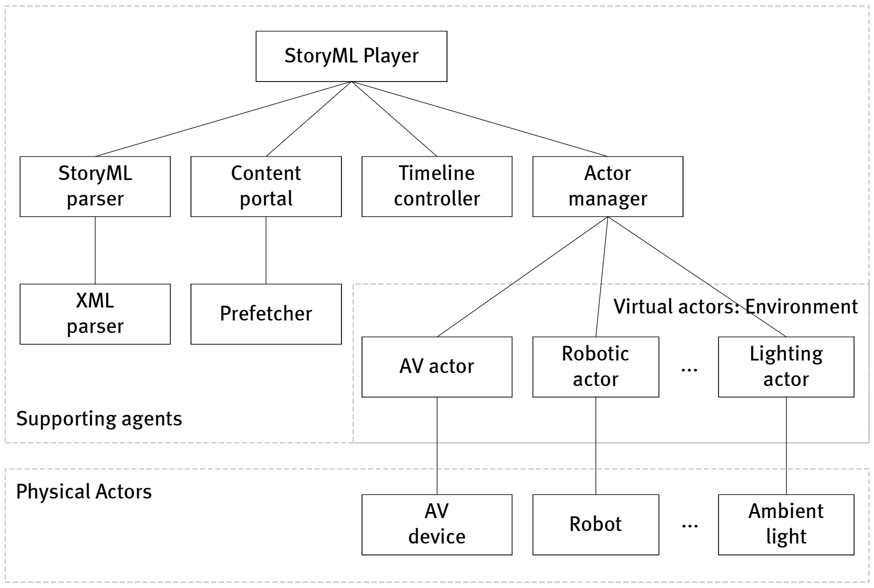


Figure 3.9: PAC based architecture of the StoryML player

For each physical agent, there is a virtual actor connected as its software counterpart. Provided with this layer of virtual actors, the system can achieve the following:

1. Decoupling of media processing from the physical interface devices and enabling process distribution. It is possible to assign media processing tasks of a physical agent, for example decoding a stream or composing a scene, to another more capable device in the network, by moving the virtual actor to that device. The processed result can then be transferred back to the physical presentation component of the physical agent for direct rendering. The media processing, therefore, can also be distributed to the network.

2. Easy switching of the user interaction from the physical device to its virtual counterpart or vice versa. The virtual actors observe and verify the availability of interface devices. If the physical environment can not satisfy the story with the preferred interface devices, the system can always provide alternatives. If a physical device is not available in the environment or the user prefers interacting with the virtual actors, then the virtual actor functions as the alternative.

3. Satisfying the requirements for the variety of the interface devices. These virtual actors can be viewed as software drivers for physical devices, which hide the differences between these devices, and provide the higher level agents with the same interface.

4. The actor manager coordinates the virtual actors by creating software agents and transferring user-events between these agents and keeps them synchronized.

3.4.4 Media and Interaction Synchronization

The XML parser analyzes a StoryML script to build a structural object representation, transferring the StoryML structure to a scheduling scheme for the presentation. This scheme is managed by a global timeline controller (figure 3.9 on the facing page) for synchronizing media objects and interactions, which can be distributed over several actors.

In StoryML, user interactions are specified by defining dialogs. These dialogs are registered to the timeline controller. At the specified moments, the timeline controller initializes a dialog by starting feed-forward media objects on target actors.

The dialog then requests the actors to listen to the user input. Unlike MPEG-4 or SMIL, StoryML does not associate any user input to a specific media object, but to an actor instead. If the user reacts, the actor will abstract the user response as an event and this event will trigger the feedback objects. If the user event results in a change in a future time, the change is registered to the timeline controller. In this way, the user interaction is synchronized.

3.5 Lessons learned from StoryML

3.5.1 Pros

The following experiences have contributed to meet the requirements (section 2.1.2) and are considered valuable to carry on to the next design cycle:

1. The StoryML player makes use of the PAC-based architecture, which emphasizes the independence of devices and the communication between the system components.

2. The abstraction component of a PAC agent is extended with DataSource and DataSink ports. A streaming channel can be built between a DataSource and a DataSink to improve the efficiency of the communication between these distributed agents while the control hierarchy of the system remains intact.

3. The StoryML player always first satisfies the desired environment described in a StoryML script with virtual software actors, which is designed as an obligatory layer in its architecture so that software actors can take places of the physical devices if they are not available. This makes mapping between the desired configuration and the actual configuration not only easier but also possible to be done on the fly if the actual configuration changes during the run time. During this mapping process, the user's preference of the configuration can also be taken into account easily.

4. An object-oriented design approach is followed, with the focus on high level structural patterns such as PAC and Streaming Channel. This approach helped the project to focus on the architectural design issues other than low-level multimedia technologies.

5. Many strengths of the StoryML system come with open technologies. The StoryML system is based on XML and Java technologies. Together, XML and Java technologies provide the StoryML system with strengths of simplicity, portability, and flexibility. XML based scripting seems to be good a direction.

3.5.2 Cons

Having said the positive things, it should also be noticed that the StoryML framework is not there yet as a generic and powerful solution for distributed and interactive media application:

1. More types of distributed interface devices should have been included for the experiment. In TOONS, although many interface devices are introduced, they were used in separate demonstrator implementations and the StoryML implementation only included two of them. Integration of these distributed interfaces needs to be done.

2. StoryML focuses on a storytelling application, and the designs of the demonstrators and their architectures are influenced by the structure of this particular type of content. For example, switching between storylines might be a good abstraction for interactive storytelling, it is certainly too limited for more complex interaction scenarios.

3. A linear narrative structure is implemented. However the design can not cover non-linear cases. An example of such non-linear structures is conditional of infinite repeating of a piece of content.

4. A timeline model is used for timing and synchronization, which requires the author to specify the exact time of every interaction dialogue. It is intuitive to use, but can be difficult if the duration of the media objects is not known in advance and if the author wants to specify timing relations such as "one after another being finished".

5. Several concepts have been introduced, such as dialogs, media objects, actors, and environments. These concepts seem not well fit in one single metaphor so that they can easily be understood by the script authors.

Nevertheless, what has been done so far not only can serve as a good starting point, but also paves the way for the final design.

CHAPTER 4

Interactive Play Markup Language

One problem of the StoryML is that it uses a mixed set of terms. "Story" and "storylines" are from narratives, "media objects" are from computer science, whereas "interactive agents" are from human computer interaction. Scripting an interactive story requires various types of background knowledge to some extent. It is questionable whether StoryML has succeeded in both keeping the scripting language at a high level and let the users, in this case the script writers, only focus on the interactive content. "Movies did not flourish until the engineers lost control to artists – or more precisely, to the communications craftsmen." (Heckel, 1991)

StoryML uses storytelling as a metaphor for weaving the interactive media objects together to present the content as an "interactive story". This metaphor made it difficult to apply StoryML to other applications when there are no explicit storylines or narratives. Moreover, StoryML can only deal with linear content structure and there is nothing but a storyline switching mechanism for interacting with the content.

Instead of StoryML, a new scripting language is needed, that has a more generic metaphor, and supports both linear and nonlinear content structures and that can deal with complex synchronization and interaction scenarios.

4.1 Play

Instead of storytelling, Interactive Play Markup Language (IPML) uses a more powerful metaphor of *play*. A *play* is a common literary form, refering both to the written works of dramatists and to the complete theatrical *performance* of such. Plays are generally performed in a *theater* by *actors*. To better communicate a unified interpretation of the text in question, productions are usually overseen by a *director*, who often puts his or her own unique interpretation on the production, by providing the actors and other stage people with a *script*. A script is a written set of directions that tell each actor what to say (*lines*) or do (*actions*) and when to say or do it (*timing*). If a play is to be performed by the actors without a director and a script from the director, the results would be unpredictable, if not chaotic.

4.1.1 Timing in a play

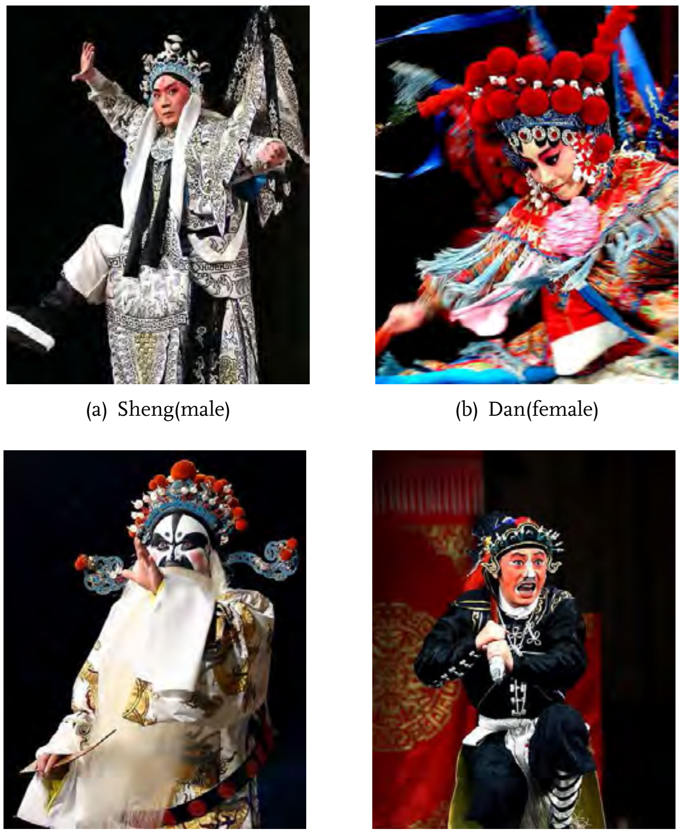
Timing in a play is very important whether it be when an actor delivers a specific line, or when a certain character enters or exits a scene. It is important for the playwright to take all of these things into consideration. The following is an example taken from *Alice in Wonderland* (Carroll and Chorpenning, 1958, p.14):

> ...
>
> **ALICE** Please! Mind what you're doing!
>
> **DUCHESS** (*tossing ALICE the baby*). Here ... you may nurse it if you like. I've got to get ready to play croquet with the Queen in the garden. (*She turns at the door.*) Bring in the soup. The house will be going any minute! (*As the DUCHESS speaks, the house starts moving. The COOK snatches up her pot and dashes into the house.*)
>
> **COOK** (*to the FROG*). Tidy up, and catch us! (*The FROG leaps about, picking up the vegetables, plate, etc.*)
>
> **ALICE** (*as the FROG works*). She said "in the garden." Will you please tell me –
>
> **FROG** There's no sort of use asking me. I'm not in the mood to talk about gardens.
>
> **ALICE** I must ask some one. What sort of people live around here?
>
> ...

A few roles are involved in this part of the play. Their lines and actions are presented by the playwright in a sequential manner, and these lines and actions are by default to be played in sequence. However, these sequential lines and actions are often not necessarily to happen immediately one after another. For example, it is not clear in the written play how much of time the duchess should take to perform the action "*tossing Alice the baby*" after Alice says "*Mind what your're doing*" and before the duchess says "*Here ... you may nurse it if you like*". The director must supervise the timing of these lines and actions for the actors to ensure the performance is right in rhythm and pace. Furthermore, things may happen in parallel – For example, the house starts moving as the duchess speaks, and Alice talks as the frog works. Parallel behaviors are often described without precise timing for performing. It is up to the directors to decide the exact timing based on their interpretation of the play. For example, the director may interpret "*As the DUCHESS speaks, the house starts moving*" as "*at the moment of the duchess start saying 'The house will be going in any minute', the house starts moving*".

4.1.2 Mapping: Assigning roles to actors

Actors play the roles that are described in the script. One of the important task of the director is to define the cast – assign the roles to actors. This is often done by studying the type of a role and the type of an actor, and finding a good match between them. This is also exactly the problem in this project for distributed presentations: determining which device or component to present certain type of media objects. It can be very hard for a computer to carry out this task, unless these types are indicated in some way otherwise.


(a) Sheng(male) (b) Dan(female)

(c) Jing (face painted) (d) Chou(clown)

Figure 4.1: Role types in Beijing opera
(Pictures from public domain)

In some traditional art of play, these types are even formalized so that a play can be easily performed with a different cast. For example, The character roles in Beijing Opera are divided into four main types according to the sex, age, social status, and profession of the character: male roles (Shēng 生, figure 4.1(a)); female roles (Dàn 旦, figure 4.1(b)); the roles with painted faces (Jìng 净, figure 4.1(c)) who are usually warriors, heroes, statesmen, or even demons; and clown (Chǒu 丑, figure 4.1(d)), a comic character that can be recognized at first sight for his special make-up (a patch of white paint on his nose). These types are then divided into more delicate subtypes, for example Dàn is divided into the following subtypes: Qīng Yī(青衣) is a woman with a strict moral code; Huā Dàn (花旦) is a vivacious young woman; Wǔ Dàn (武旦) is a woman with martial skills and Lǎo Dàn (老旦) is an elderly woman. In a script of Beijing Opera, roles are defined according to these types. An actor of Beijing Opera is often only specialized in very few subtypes. Given the types of the roles and the types of the actors, the task of assigning roles to actors becomes an easy matching game.

4.1.3 Interactive play

Plays can be interactive in many ways. The actors may decide their form of speech, gestures and movements according to the responses from the audience. Again with the example of Beijing opera, plays in early days, which sometimes can still be been seen today, may be performed in the street (figure 4.2) or in a tea house, where the actors and the audience are mixed – the actors and the audience share the stage. The movements of the actors must be adapted to the locations of the audience, and the close distance between the audience and the actors stimulates the interaction. An example of such interaction is that the characters often strike a pose on the stage, and the audience is supposed to cheer with enthusiasm. The time span of such a pose depends on the reactions of the audience. Although this is not written in the script, such an interactive behavior is by default incorporated in every play of Beijing opera.

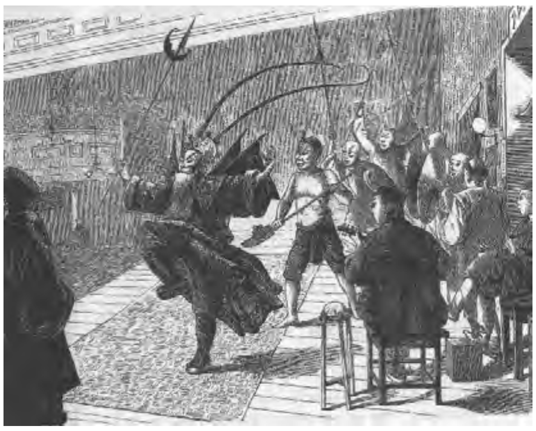


Figure 4.2: 19th century drawing of Beijing opera, public domain

Other interactive plays allow the audience to modify the course of actions in the performance of the play, and even allow the audience to participate in the performance as actors. Thus in these plays the audience has an active role. However, this does not mean that the reader of a novel, the member of audience in the theater are passive: they are quite active, but this activity remains internal.

The written text of the play is much less than the event of the play. It contains only the dialog (the words that the characters actually say), and some stage directions (the actions performed by the characters). The play as written by the playwright is merely a scenario which guides the director and actors. The phenomenon of theater is experienced in real-time. It is alive and ephemeral – unlike reading a play, experiencing a play in action is of the moment – here today, and gone tomorrow.

To clarify the discussions, the word *performance* is used to refer to the artifact the audience and the participants experience during the course of performing a script by preferred *actors*, monitored and instructed by a *director*. The *script* is the underlying content representation perceived by the authors as a composite unit, defining the temporal aspects of the performance, and containing the *actions* which are depicted

Table 4.1: Comparing theater play terms with multimedia

Theater Performance	Multimedia Presentation
Performance	Presentation
Script	Document
Content element	Media object/Media element
Action	Playback of an object/element
Actor	Player/Decoder
Director	Scheduler

by the *content elements* or the references to these elements. Traditional multimedia systems use a different set of terms which are comparable to the terms above; they are in many cases similar, but should not be confused (table 4.1).

Now that a clear view on the ingredients of script is gained, the next section first presents the basic structure of the SMIL scripting language (Ayars et al., 2005). Using the structure of SMIL and taking the extra requirements into account, the Interactive Play Markup Language (IPML) is then proposed.

4.2 SMIL

Synchronized Multimedia Integration Language (SMIL) is an XML-based language for writing interactive multimedia presentations (Ayars et al., 2005). It has easy to use timing modules for synchronizing many different media types in a presentation. SMIL 2.0 has a set of markup modules. All these modules are associated with the SMIL 2.0 namespace:

SMIL 2.0 language profile contains most of the modules in SMIL 2.0. A clear distinction between this profile and the complete set is not made.

SMIL 2.0 basic language profile is a smaller subset intended for mobile phones and other devices where computer resources are limited.

XHTML+SMIL W3C note incorporates modules from the SMIL namespace and the eXtensible HyperText Markup Language (XHTML) namespace.

These modules in SMIL 2.0 are: 1. timing and synchronization; 2. time manipulations; 3. animation; 4. content control; 5. layout; 6. linking; 7. media objects; 8. metainformation; 9. structure; 10. transitions. Not attempting to list all the elements in these modules, figure 4.3 on the next page shows an object-oriented view[1] of some basic elements: Par, and Seq from the timing and synchronization module, Layout, RootLayout, TopLayout and Region from the layout module, Area from the linking module, MediaObject from the media object module, Meta from the metainformation

[1]Note that this Unified Modeling Language (UML) model only provides a rough structure overview of the selected elements and it is not intended to replace, nor to be equivalent to, the formal DTD specification from W3C by Ayars et al. (2005).

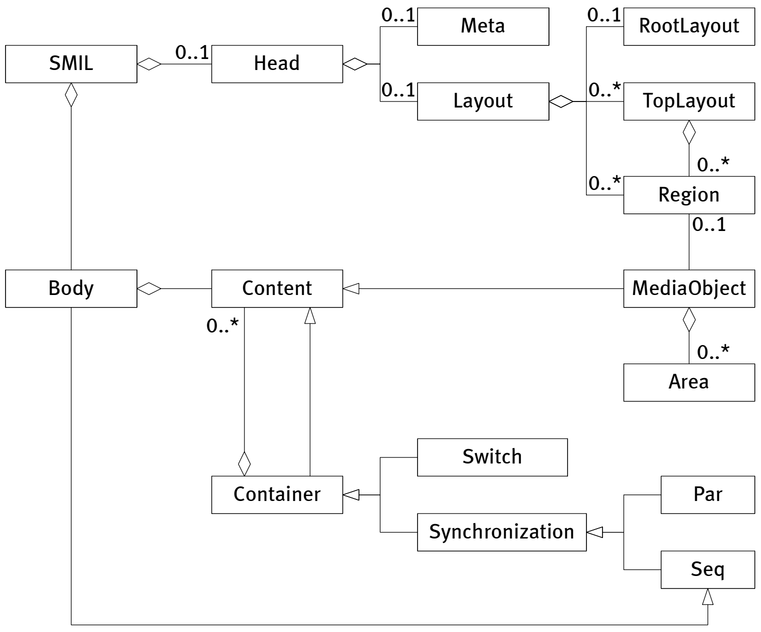


Figure 4.3: SMIL in UML

modules and Head, Body from the structure module. Details about the corresponding language elements can be found in SMIL 2.0 specification (Ayars et al., 2005), including other modules and elements that are not illustrated here.

The Region element and its attributes for two dimensional layout provide the basics for screen placement of visual media objects. The specific region element that refers to the whole presentation is the RootLayout. Common attributes, methods and relations for these two elements are placed in the superclass named the Layout.

SMIL 2.0 introduced a MultiWindowLayout module over SMIL 1.0, with which the top level presentation region can also be declared with the TopLayout element in a manner similar to the SMIL 1.0 root-layout window, except that multiple instances of the TopLayout element may occur within a single Layout element. It contains the attributes providing for creation and control of multiple top level windows on the rendering device.

Each presentation can have Head and Body elements. In the Head element one can describe common data for the presentation as whole, such as: Meta data, and Layout. All Region elements are connected to the Head. The Region elements can be connected to the Head element directly.

The Mediaobject is the basic building block of a presentation. It can have its own intrinsic duration, for example if it is a video clip or an audio fragment. The media element needs not refer to a complete video file, but may be a part of it. This is expressed in SMIL as clip-begin and clip-end attributes of the Mediaobject element. Some types of Mediaobject that have a visual component can be connected with a Region element. The lower cardinality bound for this connection on the Region side is 0 because other types of the Mediaobject may have no visual components for presentation. The connection between the Mediaobject and Region is used for describing spatial constraints in multimedia presentation. This connection gives the Mediaobject the space coordinates.

The Content, Container, and Synchronization elements are classes introduced solely for a more detail explanation of the semantics of the Par, Seq, Switch and Mediaobject, and their mutual relations.

Par and Seq are synchronization elements for grouping more than one Content. If the synchronization container is Par, it means that direct subelements can be presented simultaneously. If synchronization container is Seq, it means that direct subelements can be presented only in sequence, one at a time. The Body element is also a Seq container.

The connection between Content and Container viewed as an aggregation has a different meaning for the Synchronization element and for the Switch element. If the Container element is Switch, that means that only one subelement from a set of alternative elements should be chosen at the presentation time depending on the settings of the player. Such settings may be determined statically based on configuration settings, or they may be determined (and re-evaluated) dynamically, depending on the player implementation. If the Container is the Synchronization element, the aggregation describes subelement's timing relations.

The Mediaobject can be divided in two subclasses: TimeBasedMedia, and Static-Media, which is not shown in figure 4.3 on the facing page. The most important difference between these two classes is time clipping. The Ref, Audio, Video, Animation, and Textstream can have time clip attributes. For example, one can decide to present only two minutes starting from the second minute in a video. The Text and Img are media objects that are still.

The Area element can specify that a spatial portion of a visual object can be selected to trigger the appearance of the link's destination. The coords attribute specifies this spatial portion. In contrast, if an element is applied to a visual object, then it specifies that any visual portion of that object can be selected to trigger the link traversal. The Area element also provides for linking from non-spatial portions of the media object's display. It allows breaking up an object into temporal subparts, using attributes such as begin and end. The values of the begin and end attributes are relative to the beginning of the containing media object. The Area element can allow make a subpart of the media object the destination of a link, using these timing attributes and the id attribute (Ayars et al., 2005).

4.3 IPML

As discussed earlier in section 3.2 in chapter 3 and the previous section in this chapter, SMIL seems to have the ingredients for mapping and timing:

- Its timing and synchronization module provides versatile means to describe time dependencies (later chapter 9 will discuss timing models and compare them), which can be directly used in the IPML design without any change.

- The SMIL linking module enables non-linear structures by linking to another part in the same script or to another script. Although the Area element can only be attached to visual objects, this can be easily solved by lifting this concept up to a level that covers all the elements that need to have a linking mechanism.

- The SMIL layout module seems to be very close to the need of distribution and mapping, although it only distributes media objects among regions on the same device. But the concept of separating mapping and timing issues into two different parts, i.e. Head and Body, makes SMIL very flexible for different layouts – if a presentation need to be presented to a different layout setting, only the layout part need to be adapted and the timing relations remain intact, no matter whether this change happens before the presentation in authoring time, or during the presentation in run time.

The arguments in chapter 3 have not yet been forgotten, where SMIL is considered not directly applicable out of the shelf for the distributed and interactive storytelling: it does not support a notion of multiple devices. However later it was also found that the design went one step too far – the StoryML does incorporate the concept of multiple actors, but its linear timing model and narrative structure limited its applicability.

So what is needed to be done is to pick up SMIL again as the basis for the design of the scripting language, extending it with the metaphor of theater play, bringing in the lessons learnt from StoryML. Figure 4.4 on the next page shows the final IPML extension to SMIL. The extensions are marked gray. The Document Type Definition (DTD) of IPML can be found in background material D.

Note that in figure 4.4 on the facing page, if all gray extensions are removed, the remaining structure is exactly the same as the SMIL structure (figure 4.4 on the next page). This in an intentional design decision: IPML is designed as an extension of SMIL without overriding any original SMIL components and features, so that the compatibility is maximized. Any SMIL script should be able to be presented by a IPML player without any change. An IPML script can also be presented by a SMIL player, although the extended elements will be silently ignored. The compatibility is important, because it can reduce the cost of designing and implementing a new IPML player – the industry may pick up the IPML design and build an IPML player on top of a existing SMIL player so that most of the technologies and implementations in the SMIL player can be reused.

4.3.1 Actor

The Head part of an IPML script may contain multiple Actor elements which describe the preferred cast of actors. Each Actor element has a type attribute which defines the requirements of what this actor should be able to perform. The type attribute has a value of Uniform Resource Identifier (URI), which points to the definition of the actor type. Such a definition can be specified using for example Resource Description Framework (RDF) (McBride, 2004) and its extension Web Ontology Language (OWL). RDF is a language for representing information about resources in the World Wide Web. It is particularly intended for representing metadata about Web resources. However, by generalizing the concept of a “Web resource”, RDF can also be used to

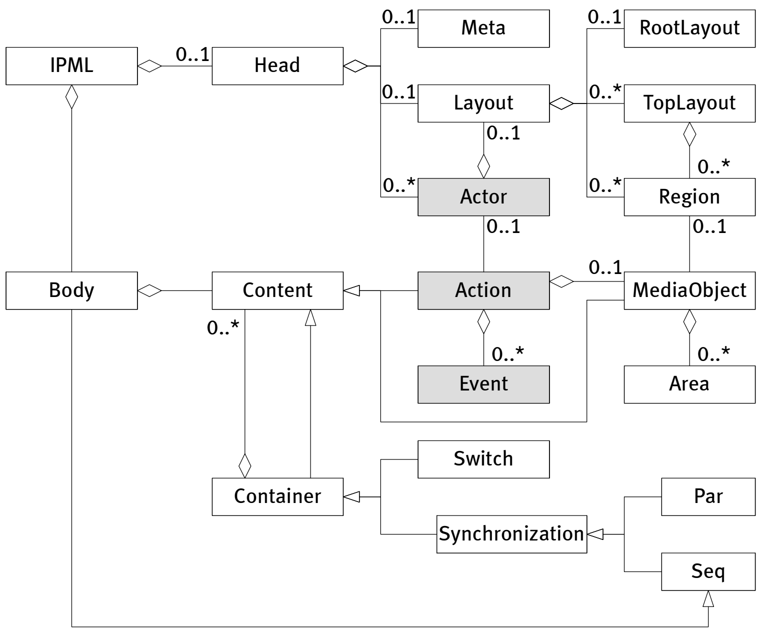


Figure 4.4: IPML in UML

represent information about things that can be identified on the Web, even when they cannot be directly retrieved on the Web. OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes. The "thing" to be described here is the type of the actor.

During the performance time, the real actors present to the theater to form a real cast. Each actor then needs to report to the director about what he can perform, i.e. his actor "type". The "type" of a real actor is defined by the actor manufacturers (well, if an actor can be manufactured). The real actor's type can again be described using an RDF or OWL specification. The director then needs to find out which real actor fits the preferred type best. The mapping game becomes a task of reasoning about these two RDF or OWL described "types". First of all the user's preferences should be considered, even if the user prefers a "naughty boy" to perform a "gentleman". Otherwise, a reasoning process should be conducted by the director, to see whether there is exactly an actor has a type that "equals to" the "gentleman", or to find an "English man" that indeed always "is a" "gentleman", or at least to find a "polite man" that "can be" a "gentleman" and that matches "better than" a "naughty boy", etc. This reasoning process can be supported by a variety of Semantic Web (Berners-Lee and Fischetti, 1999) tools, such as Python based Closed World Machine (CWM) (Berners-Lee, Hawke, and Connolly, 2004) , Java based Jena (2004) just for example. How exactly this can be done goes beyond the scope of this thesis.

4.3.2 Action

The Action element is similar to the MediaObject element in SMIL. However, Action can be applied to any type of content element which is not explicitly defined using different media objects such as Img, Video and Animation in SMIL. The Action element has an attribute src giving the URI of the content element and its type either implicitly defined by the file name extension in the URI if there is one, or explicitly defined in another attribute type. The type attribute defines the type of a content element as the type attribute of Actor defines the actor type, using a URI referring to a definition.

Action may have an attribute actor to specify the preferred actor to perform this action. If it is not specified, the type of the content element may also influence the actor mapping process: the director needs to decide which actor is the best candidate to perform this "type" of action. Again, the user preference should be taken into account first, otherwise a reasoning process should be conducted to find the "gentleman" who can nicely "open the door for the ladies".

In addition, the Action element may have an observe attribute that specifies the interested events. This attribute is designed for an actor to observe the events that are of interest during the course of performing a specific action. For example, when an actor is performing an action to present a 3D MPEG-4 object, it may be interested in the controlling events for rotating the object. This actor can then "observe" these events and react on it. Note that these observed events have no influence on the timing behavior: it will neither start nor stop presenting this 3D object, unless they are included in timing attributes, i.e, begin and end. Events that are not listed in observe will not be passed by the director to the actor during this action, therefore the event propagation overhead in a distributed network setting will be reduced.

However, actors may be interested in the events not related to any actions. To accommodate this without changing the original SMIL structure, these Actors are required to perform an action of the type null, specified using a special URI scheme "null:", which allows events to be "observed" during an action of "doing nothing".

4.3.3 Event

The third extension of IPML to SMIL is event based linking using Event elements. Event elements in an Action element are similar to Area elements in a visual MediaObject element in SMIL, with the exceptions that the parent Action element is not required to have a visual content to present, and that the events are not limited to the activation events (clicking on an image, for example). An Event has an attribute enable to include all interested events during an action, including all possible timing events and user interaction events. Once one of the specified event happens, the linking target, specified using the attribute href is triggered. Similar to the Area element, the Event element may also have begin, end and dur attributes to activate the Event only during a specified interval. Event based linking makes IPML very flexible and powerful in constructing non-linear narratives, especially for the situations where the user interaction decides the narrative directions during the performance.

4.4 Conclude with Alice in Wonderland

To show what a IPML would look like in practice, let's use the example from *Alice in Wonderland* in section 4.1.1 on page 40 again. It is impossible to embed multimedia content elements in this printed thesis, therefore two exotic URI schemes: “say:” are introduced for the lines and “do:” for the action instructions, just for the fun of it:

```
<ipml>
<head>
    <actor id="ALICE"   type="http://alice@wonderland.eu/lovelygirl" />
    <actor id="DUCHESS" type="http://alice@wonderland.eu/seriouswoman" />
    <actor id="COOK"    type="http://alice@wonderland.eu/cook" />
    <actor id="FROG"    type="http://alice@wonderland.eu/frog" />
    <actor id="HOUSE"   type="http://alice@wonderland.eu/woodenhouse" />
</head>
<body>
    <action actor="ALICE" src="say:Please! Mind what you're doing!" />
    <par>
        <action actor="DUCHESS" src="do:tossing Alice the baby"
                id="DuchessTossingBaby"/>
        <action actor="DUCHESS"
                src="say:Here...you may nurse it if you like, I've got to get
                     ready to play croquet with the Queen in the garden." />
        <action actor="ALICE" src="do:receiving the baby"
                begin="DuchessTossingBaby.babytossed"/>
    </par>
    <action actor="DUCHESS" src="do:turns at the door" />
    <action actor="DUCHESS" src="say:Bring in the soup." />
    <par>
        <action actor="HOUSE" src="do:moving" />
        <seq>
            <par>
                <action actor="DUCHESS" src="say:The house will be going
                                             any minute!" />
                <action actor="COOK"    src="do:snatches up her pot and
                                             dashes into the house" />
            </par>
            <action actor="COOK" src="do:turns to the FROG" />
            <action actor="COOK" src="say:Tidy up, and catch us!"
            <par>
                <action actor="FROG"  src="do:leaps about" />
                <action actor="FROG"  src="do:picking up the vegetables,
                                           plates, etc." />
                <action actor="ALICE" src="say:She said 'in the garden',
                                           will you please tell me—" />
            </par>
            <action actor="FROG"  src="say:There's no sort of reason asking me, I'm
                                       not in the mood to talk about gardens." />
            <action actor="ALICE" src="say:I must ask some one. What sort
                                       of people live around here?" />
        </seq>
    </par>
</body>
</ipml>
```

This then concludes the investigation into the requirements, since at this point the main architectural requirement can be summarized as:

> On top of existing network technologies and platform architectures, a generic interface architecture is to be designed to enable performing an IPML script in a networked environment with user preference and dynamic environment configurations taken into account.

The mission is clear. Now let's move on to the next part.

Part II

Architecture Design

CHAPTER 5

Pattern Oriented Architecture Design

The previous chapter introduced a language that is designed for interactive media *authoring* and that is in a set of terms of theater play. A play in such a language is basically a script with performance instructions, again, about who and when to do what. A software programmer would find it familiar to a computer program in source code – it stays static and does nothing until it is executed by the targeted system. To bring such a play alive as a performance, a *director* is needed to lead real actors to act the script out following these instructions, that is, a distributed system to *perform* the play and allowing users to interact with the play.

This chapter presents the design of the overall architecture of such a distributed system. The architecture is shaped by a collection of cooperative software design patterns, from the lower level mechanisms that cope with problems in multimedia presentation and synchronization, to the higher level structure that handles distribution, communication and user interaction issues. In this design, patterns are reusable architectural constructs that contribute to the overall architecture.

This chapter introduces the basic concepts of patterns, arguing why the "pattern language" and the Unified Modeling Language (UML) as such are insufficient for pattern specification, especially when they are applied as architectural constructs. A formal and object-oriented specification language, Object-Z, is then used to compensate the insufficiency. The discussion will also show how a general-purpose design pattern can be used as an architectural construct by presenting the Observer pattern that is frequently used at a lower level of the architecture. In the next chapters, roughly following a bottom-up approach, the patterns that contribute to the system architecture at higher levels are introduced, till an overview of the entire system architecture is reached.

5.1 Design Patterns

A common problem (or decision) in a design process, together with its best solution, is a single design pattern. The idea of capturing design ideas as a "pattern" is usually

(a) Patterns of events (b) Patterns of space

Figure 5.1: Alexander's Patterns

attributed to Christopher Alexander, an American architect. Alexander's patterns seek to provide a source of proven ideas for individuals and communities to use in constructing their living and working environments. A pattern records the design decisions taken by many builders in many places over many years in order to resolve a particular problem. Figure 5.1 shows two patterns in events and space, from his book *A Timeless Way of Building* (Alexander, 1979).

5.1.1 Software design patterns

Alexander's idea of design patterns as a problem-solving discipline had been picked up by the software architecture and design community with rapidly growing acceptance, especially after the phrase was introduced to computer science in 1995 by Gamma, Helm, Johnson, and Vlissides ("Gang of Four" or GoF in short) in their book *Design Patterns: Elements of Reusable Object-Oriented Software*. The scope of the term remained a matter of dispute into the next decade. Algorithms are not thought of as design patterns, since they solve computational problems rather than design problems. Typically, a design pattern is thought to encompass a tight interaction of a few classes and objects.

Although many of the problems frequently encountered in software design have existing solutions, the solutions can be difficult to be applied due to the need of understanding their details. Design patterns address this problem by being general; a solution documented in the format of a design pattern can be understood without involving the knowledge of the specific details. It captures key design constructs, practices and mechanisms of core competencies such as object-oriented development or fault-tolerant system design. It can speed up development processes by providing almost ready-made general-purpose solutions that have been used earlier and proven to be effective. Just like a costume sewing pattern that can be used to cut out many costumes of slightly different style, size and trim, a software design pattern is a general solution that can be tailored to fit. For example, the Observer pattern from the GoF book describes a dependency registration structure to assure consistency between objects when changes take places in one of them, by providing the core elements of a working solution, but still leaving implementation details to a particular platform and the discretion of a developer. Patterns tell the developer what to do without specifying much about how to do it. They should be abstract, yet not be vague.

5.1.2 Pattern language

To describe these general solutions in an abstract but not vague way, it is important to formalize the design decisions and the rationales behind these decisions. These decisions and rationales are often obvious with experience but difficult to document and pass on to novices. Therefore a structured method of describing good design practices within a particular domain is needed.

The *pattern language* is such a method. Alexander, Ishikawa, and Silverstein (1977) coined the term *pattern language* in the book *A Pattern Language: Towns, Buildings, Construction*. Later in his book *A Timeless Way of Building* Alexander describes what he means by pattern language and how it applies to the design and construction of buildings and towns. According to Alexander, a single entry in a pattern language should have a simple *name*, a concise description of the *problem* being addressed and the *context* in which it arises, the design tradeoffs (called *forces*) and a clear *solution*. Closely related patterns form a vocabulary, therefore serve as a language that applies to the domain of the problems they solve.

The software patterns community adopted the term pattern language and uses the meta language for documenting patterns. Several popular pattern languages exist, all of which have these basic elements defined by Alexander, but each of which adds other elements, for example *intent* and *consequences* in GoF patterns, to emphasize specific design concerns. Typically, software design patterns often include a section of *structure*, graphically illustrate the structure of the pattern. UML class diagrams and sequence diagrams are often used for this purpose, together with code fragments to illustrate how this pattern can be used in a particular programming language.

To briefly show the use of the pattern language, let's first have a close look at the Observer pattern that is frequently used in the IPML system architecture.

5.1.3 Observer pattern in pattern language

Intent

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" (Gamma et al., 1995).

Context

In an IPML performance, an actor needs to be aware of the action states of other actors in order to synchronize their behaviors, for example when another actor has just started or stopped an action. Actors also have to adapt their actions in real-time according to the reactions from the audience. In other words, when there is a state update, the actors who are interested in this update should be notified.

An actor also has to schedule the actions as soon as the timing constraints have been resolved by the director. If the actions are not to be taken immediately, they need to be planned to start or stop at a specific time. In the IPML system architecture, every actor owns, or shares with others a timer. The actor may register the actions together with the planned time and an action command (start or stop) to the timer. Once the planned time is reached, the observable timer notifies the observing actions with the registered command. A registered action may also be canceled or rescheduled if the timing constraints have been changed by user interaction or other actions.

The actors need to monitor the presenting media objects all the time. A media object first has to be prefetched to a cache so that it is ready to be started immediately. Only after it has been started, the actor may then decide to stop or pause the action at any time. The actor also should be notified when the content reaches its end. Especially when the media object is a live stream, the end time can not be predicted in advance. Obviously, the media object has state transitions of its own and these transitions are certainly interesting for the presenting actors. This is often a one-to-one relationship, but still, there is a need to separate the concerns – the actors do not know what to present beforehand and the media objects are not fixed to any actor.

Solution

The Observer design pattern has two parts and they are an *Observable* (called *Subject* in the GoF book) object and a collection of *Observer* objects. The relationship between *Observable* and *Observer* is one-to-many. In order to reuse *Observable* and *Observer* independently, their relationship should not be tightly coupled. The Observer pattern can be used in any of the following situations: 1. when the abstraction has two aspects with one dependent on the other. Encapsulating these aspects in separate objects increases the chance to reuse them. 2. when the observable object doesn't know in advance how many observer objects it has. 3. when the observable object should be able to notify its observer objects without knowing who these objects are.

1. Structure

Figure 5.2 is the UML class diagram of an observer pattern:

Observable 1. keeps track of its observers; 2. may have any number of observers; 3. provides an interface to *subscribe* and *unsubscribe* an observer object at run time; 4. sends change notifications to all the observers.

Observer provides an *update* operation to receive the notification signals from the observable object.

ObservableImpl 1. stores the observable state that is interesting for the observers; 2. serves the observers with the observable state if there is a query request.

ObserverImpl 1. maintains reference to a *ObservableImpl* object, if queries to the observable's state are to be done after the update; 2. maintains observer state; 3. implements *update* operation.

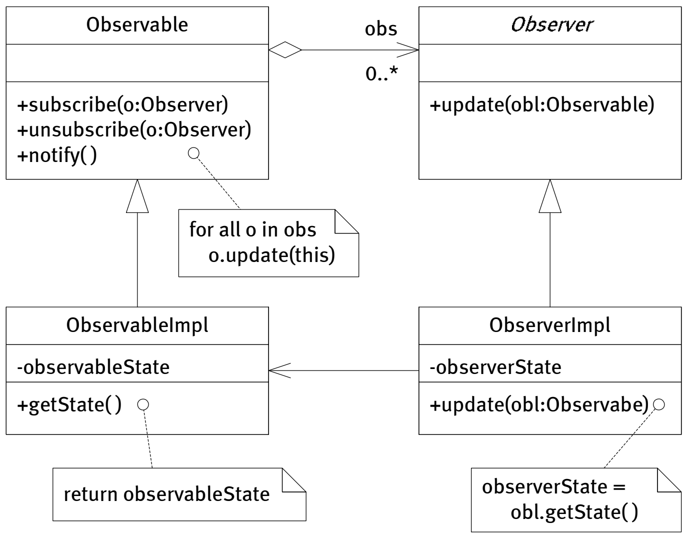


Figure 5.2: The Observer pattern

2. *Collaborations*
A UML sequence diagram or a collaboration diagram can be used here to illustrate the messaging events between the *ObservableImpl* and the *ObserverImpl* objects, which is omitted here. Nevertheless, a description is always needed to explain what is exactly happening: 1. *ObservableImpl* notifies its observers simultaneously in case of a change that could make its state inconsistent with observers. 2. after a *ObserverImpl* is notified, it queries the observable state using the *getState* function. *ObserverImpl* uses this information to change it's internal state.

Consequences

Further benefits and drawbacks of the Observer pattern include:

- Abstract coupling between the observable object and the observer objects;
- Support for broadcast communication. The notification is broadcast automatically to all interested objects that subscribed to it;
- Unexpected updates. There is a potential disadvantage of successive or repeated updates to the observers when there are series of incremental changes. If the cost of these updates is high, it may be necessary to introduce change management, so that the observers are not notified too soon or too frequently.

Code example

First an example of Java implementations of *Observable* and *Observer* is show as follows:

```
public class Observable {
  private Vector obs = new Vector();
  public synchronized void subscribe(Observer o) {
      if (o != null && !obs.contains(o))
         obs.addElement(o);
  }
  public synchronized void unsubscribe(Observer o) {
      obs.removeElement(o);
  }
  public void notify() {
     Iterator i = obs.iterator();
     while(i.hasNext())
        ((Observer)i.next()).update(this);
  }
}
public interface Observer {
  void update(Observable obl);
}
```

As an example of concrete observables and observers, a skeleton implementation of a *ObservableMediaObject* and a *MediaObjectObserver* in the IPML system is shown here, demonstrating how they cooperate to synchronize their operations according to the internal state transitions of the *MediaObject*.

```
public class ObservableMediaObject extends Observable{
  private MediaObject mo = new MediaObject();
  public SyncState getSyncState() {
    return mo.getSyncState;
  }
  public ready(){
    mo.ready();
    if(mo.getSyncState == Synchronizable.ready){
        notify();
    }
  }
  public start(){
    mo.start();
    if(mo.getSyncState == Synchronizable.started){
        notify();
    }
  }
  ...
}
public class MediaObjectObserver implements Observer {
  public void update(Observable obl) {
    if(obl instanceof MediaObject) {
      if((MediaObject)obl.getSyncState() == Synchronizable.ready) {
        //...
      }
      else if((MediaObject)obl.getSyncState() == Synchronizable.started) {
        //...
      }
      else if ...
    }
  }
}
```

ObservableMediaObject "decorates" the state control operations of a *MediaObject* (see Decorator pattern, Gamma et al. (1995); Metsker (2002)), so that the observers are updated with the change if the operation has successfully changed the synchronization state of a *MediaObject*. Once a *MediaObjectObserver* is notified, it may get the synchronization state by calling the *getSyncState* operation from the *ObservableMediaObject*.

Notice that this section is just a brief example of how the pattern language can be used for describing the design ideas in a pattern, although this "brief" is getting too long. Many details have been omitted, as well as other sections such as "known uses" that includes examples of real usages of this pattern, and "implementation issues" that provides the techniques used in implementing this pattern and suggests alternative ways for this implementation. More comprehensive descriptions can be found in the GoF book (with C++ examples) and other literature (Bruegge and Dutoit, 2004; Cooper, 1998; Grand, 2002; Metsker, 2002; Stelting and Maassen, 2002). The Observer patterns presented here and in the mentioned literature are all slightly different from each other. In practice, the Observer pattern might appear here and there without being implemented exactly the same way twice. If one invocation is used to propagate the notification event, followed by a reciprocal request for detailed state change information, the two steps can be merged into one operation by providing state information as an argument to the *notify* operation. This "push model" is not a new pattern, but rather a variant of the same pattern with "pull model".

5.2 Patterns as architectural constructs

Design patterns represent solutions to problems that arise when software is being developed in a particular context. Design patterns can be considered as reusable architectural constructs (small software frameworks) that contribute to an overall system architecture; they capture the static and dynamic structures and collaborations among key components in a software design. For example, the Observer pattern is used as the architectural foundation for distributed network sessions among actors in the IPML system. A state change in one actor usually implies state change or responses at other actors. If one actor temporarily turns off its audio presentation channel (goes mute), other actors may want to present an icon indicating that the actor is muted. If one actor leaves the system, other actors and the director must be told that they should no longer try to communicate with that actor. One can think of this use of *Observer* as an object-oriented session management framework. It can easily accommodate many types of change, such as the attendance of new actors and actor types, the way action states are computed or represented, and many others.

Object-oriented frameworks are customizable libraries of classes that can be tailored to different applications by adding subclasses that supply application-specific behavior in their method implementations. Frameworks often exhibit an Inversion of Control (IoC), or also called Dependency Injection behavior (Fayad and Schmidt, 1997), where the framework classes call application classes, in contrast to usual class libraries which are called from the application[1]. For example, a window system

[1]This behavior is similar to the Hollywood Principle: "Don't call us, we'll call you."

framework may contain the "main event loop" initiating all activity in an interactive window-based application. The framework can therefore often be seen as a skeleton application, implementing the most important design decisions at an abstract level (e.g. as abstract classes). In this respect, it is often said that a framework constitutes an overall design or architecture of a system.

Many useful frameworks, such as the Java Media Framework (JMF) (Gordon and Talley, 1998; Sullivan et al., 1998), the Java Foundation Classes (JFC) (Walrath et al., 2004), and the Apache Struts Web Application Framework (Structs) (Holmes, 2004) are rich in patterns, combing many architectural constructs in to a larger architecture. The resulting frameworks are documented by the use of known and new patterns, describing how to extend it for specific desired behaviors.

Pattern oriented system design is to use patterns as architectural constructs, build up a reusable framework, and finally reach the overall architecture of the system. There are design patterns providing the structures at different levels of the system architecture. At the bottom, design patterns such as Factory Method, Abstract Factory, Composite, Proxy, Command and Observer can be used to manage the problems in creating objects, grouping them into a bigger structure (Gamma et al., 1995; Grand, 2002). In the middle, design patterns such as Layer, Pipe, Filter and Channel control the communication among the bottom level components. At a higher level, architectural patterns such as MVC and PAC manage the problems in structuring the top level components of the system, handling user-system interaction, and making the system adaptive (Buschmann et al., 1996). This design strategy fits perfectly the requirements of the IPML system – not only a running system is needed, but also the framework underlying the system should also be reusable for future extensions.

5.3 Formal specification

5.3.1 Problems of informal specifications

The pattern language can be used for specifying the pattern oriented architecture. However, as an architectural specification language, according to Perry and Wolf (1992), it should have the following desirable properties: 1. *Generality*. The language should allow the expression of constraints at the necessary level of detail. "What is not constrained by the architect may take any form desired by the implementer". 2. *Abstraction*. Instead of "an assembly-level" architectural language that allows the specification of all conceivable configurations, yet is too verbose, a language is needed to restrict the discussion to topologies of interest. Such a language should incorporate constructions that allow for specifications that are appropriately concise. 3. *Modularity and Expressiveness*. The language is expected to provide a relatively small set of design elements from which complex expressions can be constructed.

The patterns described with the pattern language as shown in section 5.1.3 on page 55 are, at best, informal. The specification was given by means of abstract diagrams and concrete examples and by appealing to intuition. These diagrams and examples attempt to converge towards the desirable generalizations. Due to the imprecise specification, the solutions prescribed by the pattern language as such serve as prototypical abstractions of high intuitive appeal, yet they lack a more definitive foundation. To illustrate the structure of the Observer pattern, a UML class diagram is used, in which three descriptive and informal notes have to be added to explain the behaviors of three important operations (*notify*, *update* and *getState*, figure 5.2 on page 57). Otherwise, interaction diagrams, such as sequence diagrams, state machine diagrams and activity diagrams have to be employed to illustrate the dynamic behavior. However, as Booch, Rumbaugh, and Jacobson (1996, p.27) point out in an addendum to the UML reference manual,

> The interesting aspect of many patterns is their dynamic behavior. We can certainly show this with interaction diagrams, but only in the context of a specific model; it is harder to keep the "patternness" of the patterns open. Finally patterns are templates, in a sense, in which classes, associations, attributes and operations might all be mapped into different names which keeping the essence of the pattern; we need a way to clearly parameterize the patterns.

It is hard to use these diagrams to capture the roles and collaborations among components and modules of lower granularity levels. Each diagram contains constant symbols, not variables, and can depict a concrete plan, not a generic schema. In contrast, architectural specifications should express constraints on desirable properties, not the specifics of their implementation. Clearly, such expressions require variables and quantifiers as well; for example in the *notify* operation of the *Observable* class, something more formal is needed, instead of "`for all o in obs: o.notify(this)`". The java code fragments do provide possibilities to include variables to demonstrate the dynamic collaboration behavior, but at the same time, they loose even more generality and abstraction: "a collection of *observers*" is implemented as a *Vector* and it forces the specification to show the *notify* operation by traversing the elements of the Vector one by one – the information provided by the code example is redundant. A precise and concise way is needed to say "a collection of observers", and to say that the *notify* operation is to *update* all the observers. No more, no less.

Such ambiguities arise which cannot be resolved unequivocally with the pattern language as is. See for instance the confusion that resulted from the inability to conclusively pin down the difference between the patterns *Multicast* and *Observer* (Vlissides, 1997a,b). From the descriptions provided it is hard to see the difference between these two patterns – *Multicast* uses typed messages for change notifications. Also for example, the patterns mailing lists (GoF patterns, 2005; patterns-discussion, 2005), often engage in prolonged discussions whether a particular piece of code manifests an instance of one design pattern or the other, or whether one pattern is an special case of another, often without any satisfactory answer given.

5.3.2 Observer pattern in Object-Z

If, however, the underlying abstractions in pattern catalogs (collections of patterns documented in pattern language) are to become fundamental elements of software construction, they must mature at least to have a definite and unambiguous specification. Mathematical formal methods are well accepted techniques to improve the precision of software structures. Applying formal methods at early design stages can reduce ambiguities and possible errors in later software development. Moreover, formal activities such as formal verification and program generation through refinement would also bring other advantages that the UML diagrams and the natural language can not provide. This project introduces Object-Z (Duke and Rose, 2000; Smith, 2000) specification, an object-oriented extension to Z (ISO/IEC, 2002; Spivey, 1992), into the pattern language to improve its precision and conciseness, at the same time, to keep the object-oriented elementary building facilities. Again as examples, let's try to specify two basic elements of the Observer pattern in Object-Z:

$$
\begin{array}{|l}
\textit{Observable} \\
\hline
\begin{array}{|l}
\Delta \\
obs : \mathbb{P} \downarrow Observer
\end{array}
\qquad
\begin{array}{|l}
\textit{INIT} \\
\hline
obs = \varnothing
\end{array} \\
\\
\begin{array}{|l}
\textit{Subscribe} \\
\hline
o? : \downarrow Observer \\
\hline
obs' = obs \cup \{o?\}
\end{array}
\qquad
\begin{array}{|l}
\textit{Unsubscribe} \\
\hline
o? : \downarrow Observer \\
\hline
obs' = obs \setminus \{o?\}
\end{array} \\
\\
Notify \mathrel{\widehat{=}} \bigwedge o : obs \bullet [\, obl! : \{self\} \,] \mathbin{\comp} o.Update
\end{array}
$$

$$
\begin{array}{|l}
\textit{Observer} \\
\hline
\begin{array}{|l}
\textit{Update} \\
\hline
obl? : \downarrow Observable
\end{array}
\end{array}
$$

In this definition, an object of *Observable* clearly maintains a *collection*, or in other words, a set of objects of the class *Observer* and its subclasses. The polymorphic notation $\downarrow$ unites the class *Observer* and all its subclasses. The initial state (*INIT*) of an *Observable* is that *obs*, denoting the set of the subscribed *Observers*, must be empty. The state variable *obs* is declared in a Δ-list of those attributes whose values may change.

The operations *Subscribe* and *Unsubscribe* take an $o? : Observer$ as input (the decoration ? denotes an input variable), add it to or remove it from *obs* with set operations. The state variable *obs* after an operation is decorated with a single prime $'$.

The part of *Notify*, $[\, obl! : \{self\} \,]$, is equivalent to an anonymous operation

$$
\begin{array}{|l}
obl! : \downarrow Observable \\
\hline
obl! = self
\end{array}
$$

which outputs the *self* identity through variable *obl!* (the decoration ! denotes an output variable). The operation *Notify* conjoins the operation *Update* of each *Observer* in *obs*. The distributed conjunction composition $\bigwedge$ conjoins the constraints on the component operations, equating variables (including communication variables) with the same name, to model the simultaneous occurrence of the component operations. This causes all the observers $o : obs$ to be updated simultaneously. The expression $o.Update$ promotes the *Update* operation of the observer object o to the class *Observable*. The anonymous operation $[\, obl! : \{self\} \,]$ and the promoted operation $obl.Update$ are composed together with a sequential operator $\fatsemi$ – the right hand operation will proceed only after the left hand operation succeeds. The sequential operator allows communication between two operations by $\fatsemi$ equating inputs to outputs with common base names, i.e, apart from the decorations ? and !. Once a communication happens, the sequential composition hides the output from its environment.

The class *Observer* defines a skeleton operation *Update* to receive the notifying *Observable* as input, with an empty predicate part, which is the equivalent to:

$$\begin{array}{|l}
\textit{Update} \\
\hline
obl? : \downarrow Observable \\
\hline
\text{true} \\
\hline
\end{array}$$

Since the predicate part of the operation *Update* of a subclass will be conjoined, the actual result of the operation solely depends on the subclass.

The specification can be further simplified by introducing abbreviations. For example, it will be found very often that sequential composition is needed to output known values with an anonymous operation to another operation that takes these values as input, as in the definition of *Notify*: $[\, obl! : \{self\} \,] \fatsemi o.Update$. It can be simplified as $o.Update(self)$ instead, by introducing an abbreviation

$$\begin{array}{l}
Op(x_1 \leadsto v_1;\ x_2 \leadsto v_2;\ \ldots;\ x_n \leadsto v_n) == \\
\qquad ([\, x_1! : \{v_1\};\ x_1! : \{v_2\};\ \ldots;\ x_n! : \{v_n\} \,] \fatsemi Op)
\end{array}$$

given that *Op* is any operation that has input variables of compatible types with the same base names: $Op \mathrel{\widehat=} [\, x_1? : X_1;\ x_2? : X_2;\ \ldots;\ x_n? : X_n;\ \ldots \mid \ldots \,]$ where $v_i \in X_i$ $(i = 1 \mathrel{..} n)$, and when $x_1?$ is the only input variable in the operation *Op*. Further it can be simplified it as $Op(v_1) == ([\, x_1! : \{v_1\} \,] \fatsemi Op)$.

Let's also introduce an abbreviation ":=" for simple operations that update a state variable x of type X with a variable v of the same type:

$$x := v == [\, \Delta(x) \mid x' = v \land v \in X \,]$$

so that the assignment operations can be written in a more readable form.

By introducing these abbreviations and by omitting derivable variable declarations, the class Observer can be rewritten as:

$$\begin{array}{|l}
\textit{Observable} \\
\hline
\begin{array}{|l|}
\hline
obs : \mathbb{P} \downarrow Observer \\
\hline
\end{array}
\quad
\begin{array}{|l}
\textit{INIT} \\
\hline
obs = \varnothing \\
\hline
\end{array} \\
\\
Subscribe \mathrel{\widehat=} [\, o? : \downarrow Observer \,] \bullet obs := obs \cup \{o?\} \\
Unsubscribe \mathrel{\widehat=} [\, o? : \downarrow Observer \,] \bullet obs := obs \setminus \{o?\} \\
Notify \mathrel{\widehat=} \bigwedge o : obs \bullet o.Update(self) \\
\hline
\end{array}$$

5.3.3 Benefits of formal specification

Comparing to the UML diagram and the code example, the "collection of observers" and the operations on this collection are modeled as a set, which provides the abstraction that is no more and also no less than is necessary, yet has firm and precise semantics. The informal note "`for all o in obs: o.update(this)`" in the diagram has now a formal and concrete body, yet remains at an abstract level that the Java sample code can not achieve. Notice that there is no self-reference in UML, "*this*" is just a Java or C++ convention, whereas in Object-Z it is defined in the language with the *self* identity. While it is hard to ensure the correctness, and even the syntactical correctness of the UML diagrams and the additional notes in natural languages, the specification in this chapter is checked with Wizard, a type-checker for Object-Z (Johnston, 1996).In a design process Object-Z can be used to derive object-oriented architectures from abstract functional specifications, where refactoring rules such as *annealing*, *coalescence* and *reflection* can be applied (McComb, 2004; McComb and Smith, 2004), and correctness concerns can be addressed as part of the design process (Derrick and Boiten, 2001; Smith, 2000).

In short, the Object-Z specification provides a better amount of generality and abstraction than UML and concrete examples, with the power of both formalization and object orientation for elementary element building blocks – it has the properties that an architectural specification should have. However, the Object-Z specification is not intended to replace natural language specifications. Instead, formal definitions are complementary to existing means (natural language, code samples, etc.). By definition, descriptions in natural language cannot always be conclusively and fully formalized. One should be aware that in some cases formalization is difficult and that there is a definite limit to what can be specified in Object-Z. Object-Z is used only to treat the solution part of the pattern specifications (more specifically, the architectural construct specified by the solution). It has not been found that there is any attempt to formalize the *intent*, *context* and *consequences*. This does not mean, however, that these elements in the specification of patterns are insignificant. The static structure and the dynamic behavior of a pattern might be the first, and probably the easiest part to start with. Here let's leave the possibility of formalizing the other parts to future research.

This chapter has clarified the preliminary concepts of using software patterns as architectural constructs and the need of employing Object-Z specification to improve the precision and concision in addition to UML diagrams. It is now the time to describe the basic problems in the IPML system and the pattern solutions thereof, as well as later the communication middleware and the overall architecture.

CHAPTER 6

Actions

At the bottom level of the IPML system the actors are acting, i.e., performing their actions on the media objects. This chapter describes three patterns applied at this level. Timed Action is a new pattern, designed for asynchronous control of timed behavior of the actors; Synchronizable Object pattern lifts the structure of active media objects seen in many multimedia systems to a pattern level, to deal with the state control of heterogenous media objects; Action Service Factory pattern combines the Timed Action and Synchronizable Object with the generic Abstract Factory pattern (Gamma et al., 1995), together providing action services transparently.

6.1 Timed Action

6.1.1 Intent

The Timed Action pattern decouples action execution from action invocation to improve concurrency so that the action can be taken asynchronously in another process, and possibly at a preferred future time. It also simplifies synchronized access to the action service supplier that has its own thread of process in parallel.

6.1.2 Context and forces

In order to synchronize media elements, many multimedia systems implement a central scheduler to control the timing behavior of every element in sync by sending action requests (for example starting, stopping and pausing) directly in real time, expecting these operations to be immediately executed by the receiving media elements. This strategy works fine if, and only if

1. the central scheduler can indeed schedule every timing behavior of all the involved players and elements in advance, and

2. the signal will be received by the targeted elements instantly, or in other words, in a time that is short enough to be ignored.

However neither holds in an interactive and distributed context. For example in the IPML system, the timing of an action, or the operations on the media objects may depend on local situations (sensorial input from the environment) and user interaction. In this case, the execution time can not be scheduled in advance by the central scheduler (the director) because the operations can not be pinned down at absolute time points in advance. Scheduling has to be done in real time. Then there will be two options, either schedule the actions centrally by the director, or partially by the actor themselves but supervised and coordinated by the director. Indeed the actors can send every local sensorial input and the interaction events to the director and let it decide what to do next on behalf of the actors in real time. However, not every decision of an actor should be taken care of by the director – the director is not there for that purpose. This approach would heavily increase the network load, worse, as the number of actors increases, the central scheduler would also be overloaded with trivial decisions that the actors can make by themselves locally.

Even when the director is able to handle this load, there is no guarantee that in a network with heterogenous connections, action requests will arrive at actors and will take effect on media objects in an ignorable short time. To ensure actors will act at right moments, action requests have to be received by the actors prior to the expected effecting time – this is possible because many of the timing constraints can be resolved earlier than the actual action should be taken. In the worst case, when some of the action requests can not arrive at the destination before the expected time, a network delay can be taken into account. It is then possible to ensure that all the involved actors will wait for the period of the delay, compromising the timing accuracy for better synchronization if the operations are intended to be done at the same time. In both cases, actions that arrive earlier need bo be hold until the right moment.

Often, media objects have their own thread of control, for example, a MPEG-2 video stream with hardware decoding. Certain actions on this stream should be synchronized, for example *Start* and *Stop* should not happen at the same time on the same stream otherwise it might result in an undetermined finishing state. For those media objects which do not have an internal mechanism to guarantee the synchronized access to these actions, an external one is needed.

Further, concurrent operations on the media objects should not block the entire process. For example, although *Start* and *Stop* are not allowed to happen at the same time on the same media object, *Stop* should still be possible to be queued up while another process is *Start*ing the element in order to *Stop* it immediately after it is *Start*ed.

In short, in the IPML system, there is a need for actors and media objects to schedule the actions locally and to handle synchronized action requests in parallel. In a more generic sense, many other applications may also benefit from a structure that has it own process for providing the operation services, instead of executing these operation in the processes of the requesting clients. However, if multiple clients are allowed to send the operation requests, synchronized accesses to the state and the operations of the serving object must be guaranteed, such that the accesses from different processes are mutually exclusive.

6.1.3 Solution

Every action service supplier decouples action invocation from action execution, so that the client process appears to invoke an ordinary function call with a parameter. The parameter specifies expected execution time, and a list of other parameters as well. The function call is automatically converted into a correspondent scheduling object and registered to a scheduler that has a process of its own, checking whether the scheduling objects have reached their planned execution times. Once a scheduling object has reached its planned time, the scheduler sends it back to the action service for execution, where the scheduled object is converted back to a function call and executed in the process of the action service.

The Timed Action pattern consists of the following components. An *ActionService* implements and executes the action operations. For the *ActionService*, an *ActionServiceProxy* represents the action interface as direct operations. Upon request, it transforms the client's action invocation into a *TimedAction* object, subscribes it to a *Scheduler*, and returns a *TentativeResult* to the client. The *TimedAction* object wraps an *Action* object and its planned execution time together, in which the *Action* object encapsulates the serviced action operations from the *ActionService*. The *Scheduler* then activates the *TimedAction* at the planed time. Once activated, the *TimedAction* object enqueues its *Action* component to an *ExecutionQueue* object that has its own active process to dequeue the actions, execute them and update the *TentativeResult* object with the actual final result.

6.1.4 Structure

The overall structure of the Timed Action pattern is illustrated in figure 6.1 on the following page using a UML class diagram. See background material E for detailed Object-Z specifications of these components.

6.1.5 Consequences

Benefits

The Timed Action pattern provides the following benefits:

Action scheduling: Actions invoked asynchronously are executed based on the time constraints specified by the client. The order of action execution is independent from the action invocation.

Enhanced concurrency: Concurrency is enhanced by allowing client processes, action scheduling and action execution to run simultaneously.

Simplified synchronization: Synchronization complexity is simplified by the *ExecutionQueue*, which guarantees serialized access to *Action* execution and as a result serialized access to operations in *ActionService*.

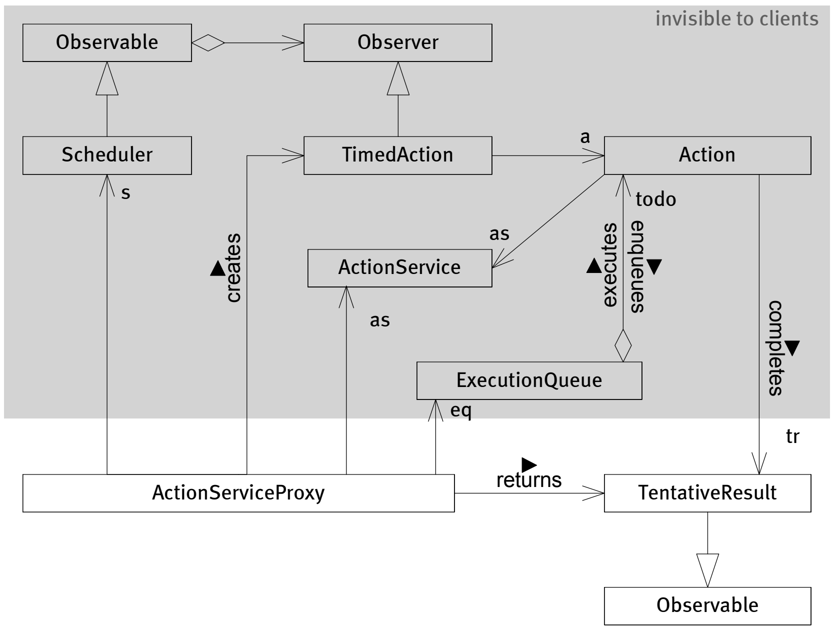


Figure 6.1: Timed Action pattern

Encapsulated scheduling policy: The scheduling policy is encapsulated in its own class and can be changed easily. For example, the class *Scheduler* may maintain the *TimedAction* objects in a sequence instead of a set, such that the order of action requests can be taken into account when multiple *TimedAction*s are planned to be executed at the same time and if there is a need of notifying these objects in sequence on a first-come-first-serve basis.

Liabilities

However the Timed Action pattern has the following liabilities:

Complexity and performance overhead: The Timed Action pattern involves multiple parties working together in parallel to accomplish a single action request, which certainly increases the context switching, synchronization and data movement overhead. If there is no need of planning an action for future execution, the Active Object pattern (Lavender and Schmidt, 1996) can do the job without employing an extra *Scheduler* for TimedActions. If it is only to ensure one action at a time to be executed within a passive object, the Monitor pattern (Buschmann et al., 1996) incurs even less context switching and data movement overhead.

6.2 Synchronizable Object

6.2.1 Intent

The Synchronizable Object pattern extracts the common inter-media synchronization behavior from the media objects with intrinsic timing and the ones without, such that different media objects can be synchronized at distinct and possibly user definable synchronization points.

6.2.2 Context and forces

A multimedia system uses multiple different media (text, audio, graphics, animation, video, and in an ambient intelligent environment, movements and behaviors of physical objects), to convey information in synchronized order and at synchronized moments. Many media objects have intrinsic timing (audio, video, animation), and some don't but require the content to be presented in a given order (speech and movements), where as the others are static (text, image and graphics). Some media appear active (video, audio, animation and Text-to-Speech (TTS)) and have a automatic and successive behavior, whereas some others are passive (presentation slides, linked web pages) and require external drive to move forward. How to synchronize all these different types of media is not trivial. Especially in an ambient intelligent environment, media are considered in a generic sense, that is, any physical or virtual object that conveys information. Designers have the freedom to create new forms of media hence new media types should be able to be incorporated into the system easily.

In Presentation Environment for Multimedia Objects (PREMO) (Duke, Herman, and Marshall, 1999) and StoryML(Hu, 2001, 2002, 2003), all the media objects are assumed to be active, that is, every object has its own process driven by a *Timer* (PREMO) or a *MediaClock* (StoryML), even if the object does not have intrinsic timing. This approach simplifies the synchronization by providing a common time-based mechanism for all the objects, however may increase the process resource overload caused by the unnecessary active process for objects that need not to be actually active.

In JMF(Gordon and Talley, 1998; Sullivan et al., 1998), timed media objects or streaming media objects that have intrinsic timing, for example, audio and video objects, are treated differently from media objects that do not have intrinsic timing. While synchronization mechanisms (Libraries and APIs) are well defined for timed media objects, other media objects such as graphics and text are treated as different data types for which no synchronization mechanisms are defined. To synchronize with the timed media objects, the controlling interfaces of streaming media have to be implemented on top of these non-timed types, which is not only unnatural, but also sometimes hard to impose the semantics of timed media on the non-timed ones. For example, should an *EndOfMediaEvent* be triggered when a picture has been presented? or never?

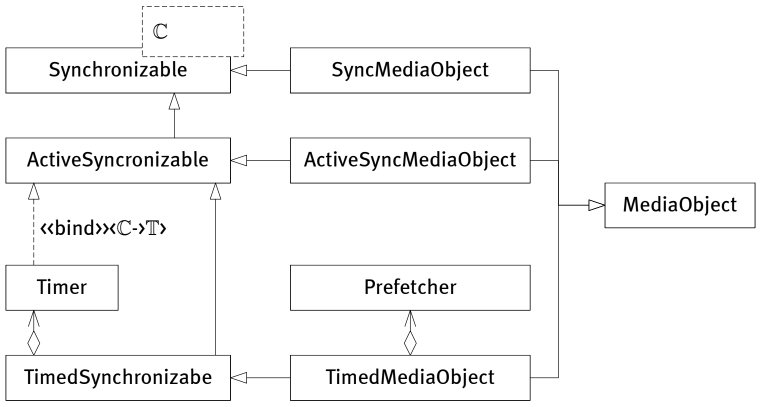


Figure 6.2: Synchronizable Objects

6.2.3 Solution

The design of this pattern is very much inspired by the structure appeared in many multimedia standards and systems, especially JMF and PREMO. The formal specification of the synchronization objects is based on the foundational objects described by Duke et al. (1997), omitting exception handling mechanisms that are considered not to be important in a pattern specification, and applying a different event handling design. The design is then extended with the notion of time and the controlling structure inspired by JMF.

6.2.4 Structure

As the basis, the concept of the coordinate system is introduced. Every *Synchronizable* object has a progressive behavior – to be started, step forward until the end if there is an end, and then possibly repeat from the beginning. A step is modeled as a coordinate point and the sequence of the steps forms a coordinate system. An *ActiveSynchronizable* object automatically steps forward in its own process and a *TimedSynchronizable* automatically moves forward but also with timing constraints. The timing constraints are applied to the *TimedSynchronizable* by synchronizing with a *Timer*, which itself is an *ActiveSynchronizable* with a moving-forward or a moving-backward timeline at a specific speed. Synchronizable media objects are Synchronizable objects at all the levels of the inheritance hierarchy, but at the same time implement the presentation behavior as a *MediaObject*. In addition, a *TimedMediaObjects* needs a *Prefetcher* that keeps fetching enough amount of data ahead of the presentation process, so that immediate starting and continuous presentation are possible. Figure 6.2 shows the hierarchy of the synchronizable objects.

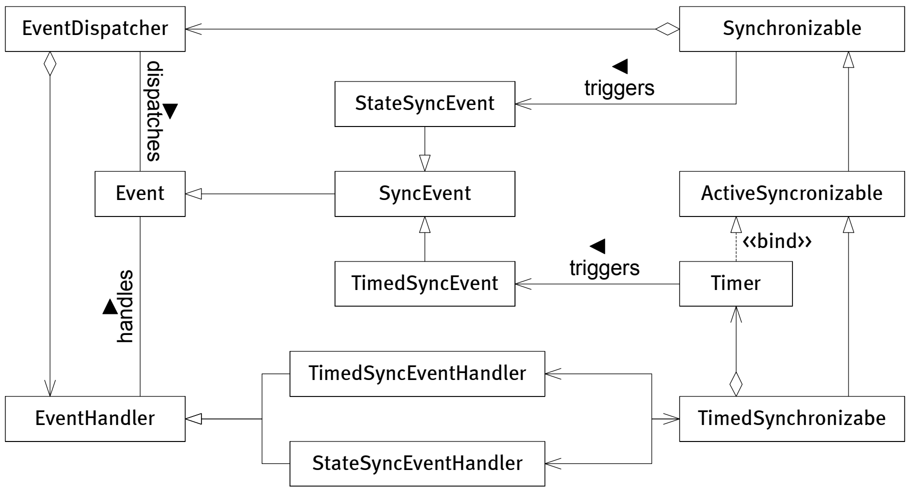


Figure 6.3: Event based synchronization

The synchronization between *Synchronizable* objects is event driven. A *Synchronizable* may trigger *StateSyncEvents* when it changes its synchronization state (stopped, ready, started and paused), may also issue other events that are attached to a coordinate when the coordinate has been passed. Every synchronizable object uses the Reactor pattern (Schmidt, Stal, Rohnert, and Buschmann, 2000) to dispatch the synchronization events to the event handlers that are interested in certain sets of events and sources. They an object of the type *EventDispatcher* that selects the registered *EventHandler* objects and dispatches the synchronization events of the interests. The *EventHandler* object in turn invokes concrete event handling operations in other objects. The *TimedSynchronizable* object synchronizes the internal state with a timer (reacting on the *StateSyncEvents*), and present the data on the paces of that timer (reacting on the *TimedSyncEvents*), which also demonstrates the use of this event-driven approach. See figure 6.3 for the UML class diagram that shows the static structure of the synchronization events. Also see background material F for detailed Object-Z specifications.

6.2.5 Consequences

Benefits

The Synchronizable Object pattern offers the following benefits (please refer to background material F for the details mentioned here):

Separated timing concerns: Timing constraints are added only when they are necessary and the timer can be shared – which saves concurrent processing resources. Static media objects, for example, static pictures, can be easily implemented as passive *Synchronizable* objects with a simple coordinate system that only has one coordinate and it will stay static until there is an external drive to move it forward – which naturally stops it because there is no next coordinate. If there is a need for the picture to stay presenting itself for a certain amount of time, it can be then implemented as an *TimedSynchronizable* that uses a timer. For media objects that have a progressive presenting behavior and that are not constrained by the time, for example, a TTS object, the *ActiveMediaObject* class does what is needed. However, if the TTS object need to presented at a speed of a video stream, a timer can then be shared to make a *TimedMediaObject* so that the TTS object can speak faster or slower.

Flexible concurrency: The concurrency of the structure can be from none to fully concurrent. A *Synchronizable* can be moved forward by an external process and the events can be dispatched in the same process using the Reactor pattern (Schmidt et al., 2000). The concurrency can be improved by:

1. implementing a stepping process of its own, asynchronously presenting the corresponding data;
2. incorporating a timer process that keeps track of the time and drives the stepping process forward or backward at a preferred speed;
3. using an independent *Prefetcher* process that keeps fetching the data from the source ahead of the presentation process to ensure immediate start and continuous presentation;
4. applying the Proactor pattern (Schmidt et al., 2000) to involve an active process for asynchronous event dispatching.
5. introducing an active *EventDispatcher* for each synchronizable object.

Simple and unified synchronization interface: All the *Synchronizable* objects at different levels have the same synchronization controlling interface *Ready*, *Start*, *Stop* and *Pause* and have the same state transition scheme, no matter whether the coordinate system is time based or not. This makes it easy to incorporate new types of *Synchronizable* objects without influencing existing types.

Open for extension: The actual presentation behaviors of media objects are left open, and the event-based synchronization mechanism can be extended for other purposes. For example, user interaction events can be easily added to the structure in addition to the state transition events and the timing events.

Liabilities

Hard to debug: When multiple processes are involved, a *Syncronizable* object can be hard to debug since the inverted flow of control oscillates between the framing infrastructure and the parallel callback operations on the synchronization controls and event handlers. This increases the difficulty of "stepping-through" the runtime behavior of the object with a debugger, unless the debugger efficiently supports multi-thread tracking, and the developer fully understands the structure of event-handling and synchronization.

Event dispatching efficiency depends on the hosting platform: This is a liability of all event based synchronization mechanisms. In the specification, events are sometimes required to be dispatched simultaneously using a conjunction operator (for example, the operation *DispatchSyncEvent* of *Synchronizable* and the operation *Dispatch* of *EventDispatcher*. This can only be applied efficiently if the hosting platform and the operating system supports efficient parallel processing. It is possible to emulate the semantics of the simultaneous operation using multiple threads, e.g., one thread for each composed operation, then serialize the threads to a single process. However this can only be efficient when the number of the included threads is small, and each thread does not block the entire process and impede the responsiveness.

6.3 Action Service Factory

6.3.1 Intent

To provide a contract to create families of related or dependent objects for performing synchronization tasks on a given source of media content, without having to specify the concrete classes and without knowing how to create these objects.

6.3.2 Context and forces

The IPML actors need to perform actions on the media contents and these contents vary in types. Given a source of media, the actor should be able to determine its type and create a synchronizable media object for manipulation if the actor is capable of doing so. For an extensible architecture, the system can not assume the concrete capability of an actor, but it does require the actors to report their capabilities and to create the media objects of the types within their reported capabilities.

Furthermore, these synchronization actions need to be served across the network between the actors and to be scheduled by the director for instant or future execution. The Timed Action pattern can be applied, however, the media object as well as the related synchronization actions must then be adapted to provide action service. A set of objects, for example, event dispatchers for synchronizable objects and local schedulers for timed actions must be created altogether.

These related or dependent objects can be created in different ways in order to incorporate specific event dispatching and task scheduling strategies depending on the available software and hardware resources and the implementation. Again, the system can not assume the concrete strategies, but it does require the actor to be able to create all related or dependent objects, assemble them together and provide simple interfaces at an abstract level.

6.3.3 Solution

Both the Factory Method pattern and the Abstract Factory pattern (Gamma et al., 1995; Metsker, 2002) are applied. The classes *AbstractSyncFactory* and *AbstractSyncServiceFactory* are abstract factories with several abstract or concrete Factory Methods.

The concrete Factory methods delivers the required products. The class *AbstractSyncFactory* has a concrete Factory Method *CreateSynchronizable* that takes a media source as input and delivers a *Synchronizable* object. The class *AbstractSyncServiceFactory* has a concrete Factory Method *CreateSyncServiceFactory* that in turn takes the *Synchronizable* object as input and delivers a *SyncServiceProxy* object that serves the timed synchronization actions.

Both abstract factories also have several abstract Factory Methods with only the input and output interfaces defined. These abstract Factory Methods, for example *CreateEventDispatcher* and $CreateSynchronizable_0$ from *AbstractSyncFactory*, and *CreateScheduler* from *AbstractSyncServiceFactory*, are to be implemented in the concrete factories.

The classes *ActionService* and *ActionServiceProxy* from the Timed Action pattern are used as templates to create *SyncService* and *SyncServiceProxy* by binding specific synchronization operations (*Ready*, *Start*, ..., etc.) to the template operations (Op_1, Op_2, ..., etc.) Using patterns as templates is a typical way of applying patterns. In Object-Z, Template binding can be easily formalized as replacing (renaming) the state variables and operations from the template class.

6.3.4 Structure

The overall static structure is shown in figure 6.4 on the next page. What is not shown in figure 6.4, is that the actors are supposed to provide the concrete factories. The actor may also implement multiple concrete factories and decide which concrete factories to be used at the initialization time, or according to the changes in the system, change the concrete factories to different ones. The director and other actors only need to see the same abstract factories – the interfaces for the creation of the media objects and action service proxies. See background material G for detailed Object-Z specifications.

6.3.5 Consequences

Please refer to background material G for certain details mentioned in the following discussions.

Benefits

The Action Service Factory pattern offers the following benefits:

Flexibility: The Action Service Factory pattern increases the overall flexibility of the architecture. This flexibility manifests itself both during the design time and the runtime. During design, it is not necessary to predict which media

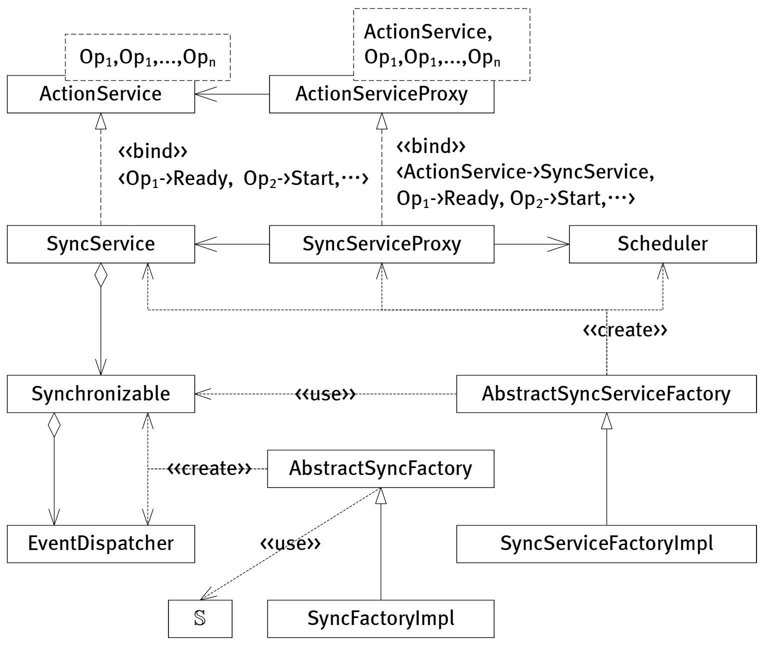


Figure 6.4: Action Service Factory

types an actor can perform and how corresponding media objects will be created and served. Instead, one can focus on the generic framework and then develop implementations independently for each involved actor later. It can also simplify the testing of the rest of the application. Implementing testing factories and products are simple; it can simulate the expected resource behavior. At runtime, the system can easily integrate new actors and the actors can easily integrate new capabilities when features and resources are available.

Reusability: The clients of these factories are independent of how the *Synchronizalbe* objects and the service accessing proxies are created. On the one hand, the Action Service Factory pattern makes it easy to create and manipulate these objects without knowing exactly what they are. On the other hand, following the protocol of the abstract factories, one can design and implement concrete factories and products that operate in diverse environments. Hence the anonymity of the concrete factories and products promotes reuse – the code that uses these objects doesn't need to be modified if the factories produce instantiations of different media objects and proxies than they used to.

Compatibility in created objects: By forcing the other objects to go through the interfaces *CreateSynchronizalbe* and *CreateSyncServiceProxy* to create concrete *Synchronizable* objects and service accessing proxies, the Action Service Factory pattern ensures that the client objects uses a compatible set of objects.

Liabilities

Since the clients of the factories do not see the concrete products (concrete media objects and their proxies), but only the abstract products (*Synchronizable* objects and their proxies), the synchronization interfaces of these abstract products are vital and should be carefully designed. If these interfaces are improperly defined, producing some of the desired media objects can be difficult or even impossible. In this case, the interfaces of the abstract products must be redesigned to incorporate the new synchronization behavior, which can be a lot of work to update all related concrete products.

6.4 Concluding remarks

The patterns of this chapter deal with the essentials of media processing: timing, event handling, synchronization, presentation, etc. If a single-device architecture for media processing would have to be designed, the above patterns would be sufficient. But the typical target architecture is distributed and heterogeneous. This puts extra demands both to the architecture and to the methodology of patterns. These topics are addressed in the next chapter.

CHAPTER 7

Communication

Software components residing in different systems can communicate with one another over a network in many different ways. At a low level transport layer, techniques such as socket mechanisms (Stevens et al., 2003) can be directly used, together with a custom communication protocol to serve the communication needs. However, it leaves many complex problems completely to the implementation of the custom protocol. The problems are for example the quality of communication, the guarantee of security, the management of heterogeneity, and the control of concurrency. These problems can not be ignored when a distributed real-time multimedia system is to be designed. A variety of higher level communication protocols and frameworks, from the synchronous Remote Procedure Call (RPC) to asynchronous data channels, can be the better choices for the purpose.

The patterns described in this chapter deal with the communication issues in a distributed architecture, with attention paid to the management of concurrency, heterogeneity, quality of service, and the load of media data transportation.

7.1 Overview of communication patterns

7.1.1 Remote Procedure Call

In this model, a component acts as a client when it requests some service from another component that acts as a server when it responds to the request from the client. RPC makes invoking an external procedure that resides in a different system or machine in the network almost as simple as calling a local procedure. Arguments and return values are automatically packaged in a platform-neutral format and sent between the local and remote procedures. For each remote procedure, the underlying RPC framework needs a “stub” procedure on the client side acting as a proxy for the remote procedure, and a similar object on the server side acting as a proxy for the client. The role of the stub is to take the parameters passed in a regular local procedure call and pass them to the RPC system that must be resident at both the client and the server sides. Behind the curtain, the RPC system cooperates with the stubs on both sides to transfer the arguments and return values over the network. RPC frameworks have become an established technique, and are typically invisible to application programmers since they merely represent the basic transport mechanism used by more general middleware platforms as presented in section 7.1.4.

An example of the RPC system is Open Network Computing (ONC) from Sun Microsystems (Srinivasan, 1995) for UNIX platforms. It is used for accessing system services such as the Network File System (NFS) and the Network Information System (NIS) (Stokely and Clayton, 2001). Another example is the Distributed Computing Environment/Remote Procedure Calls (DCE/RPC), which is used as the basis for the Microsoft COM+ (COM+) middleware (Eddon and Eddon, 2000; Pinnock, 1998).

7.1.2 XML-based RPC

Although the issue of interoperability in networked systems is addressed explicitly by RPC, client and server components are tied to the same RPC framework in order to cooperate successfully. Every RPC system has a different data structure and encoding system, and it is hard to communicate if the client and the server are operating on a different RPC dialect. Fortunately, the basic semantics of most RPC systems are quite similar since all of them are based on a synchronous procedure call in a C-like syntax format. The solution is to develop a common language that is understandable by both parties, even though they are running on different RPC systems.

The recent development on this is to treat the messages sent between clients and servers as XML documents with a given syntax. Using XML to define the syntax of RPC requests and replies was a simple idea, but it paved the way for interoperability between different RPC systems (Hunter et al., 2004; Laurent, Dumbill, and Johnston, 2001). The essential idea in XML-based RPC frameworks is to use XML to define a type system for data communication between clients and servers. These type systems specify primitive types such as integers, floating points, and text strings, and they provide mechanisms for aggregating instances of these primitive types into compound types in order to specify and represent new data types.

One of the XML-based RPC frameworks was the Simple Object Access Protocol (SOAP) (Gudgin et al., 2003; Mueller, 2001). SOAP was proposed to W3C by HP, IBM, Microsoft, SAP and many other companies. There are several different types of messaging patterns in SOAP, but by far the most common is the RPC pattern, where one network node (the client) sends a request message to another node (the server), and the server immediately sends a response message to the client. An advantage of SOAP is its extensibility by the use of XML schemas and the fact that the widespread HyperText Transfer Protocol (HTTP) protocol can be used as the transport mechanism between clients and servers, thus using the Web as a communication infrastructure and a tunnel between cooperating distributed applications. HTTP was chosen as the primary application layer protocol for SOAP since it works well with today's Internet infrastructure; specifically, SOAP works well with network firewalls. This is a major advantage over other distributed protocols like General Inter-ORB Protocol (GIOP), Internet Inter-ORB Protocol (IIOP) or Distributed Component Object Model (DCOM) which are normally filtered by firewalls.

Other XML-based RPC dialects such as standard XML-RPC have also been specified (Laurent et al., 2001; Winer, 1999). Most of them, however, do not incorporate XML schemas and are thus limited to a fixed set of primitive data types, as are traditional RPC systems. Fortunately, the problem of having to cope with different dialects of XML-based RPC systems is not as serious as dealing with incompatible classical RPC frameworks, because in many situations simple transformation rules can be defined to convert XML encoded messages between different RPC platforms.

7.1.3 Remote Method Invocation

Remote Method Invocation (RMI) is similar to RPC, but integrates the distributed object model into the Java language in a natural way (Grosso, 2001; Sun Microsystems, 2003). With RMI, it is not necessary to describe the methods of remote objects in a separate type definition file. Instead, RMI works directly from existing objects, providing seamless integration. Remote objects can be passed as parameters in remote method calls, which is a feature that classical RPC does not possess.

In classical RPC systems, client-side stub must be generated and linked into a client before a remote procedure call can be made. In RMI systems, a local surrogate (stub) object manages the invocation on a remote object. The stub that is needed for an invocation can be downloaded at runtime (in an architecture- neutral bytecode format) from a remote location, for example directly from the server just before the remote method is actually invoked. RMI seems to fit well with the general trend of distributed systems becoming increasingly dynamic.

7.1.4 Asynchronous Protocols

Although the interoperability of RPC can be enhanced with XML-based communication and object oriented technologies, the communication using these models is basically synchronous: the client is blocked while the call is processed by the server. To enable asynchronous communication, multi-threading is required. Multi-threading is more difficult to handle and is more error prone, however distributed systems, especially, distributed multimedia systems often require multi-threading for parallel media presentations and at the same time user interaction. Even when multi-threading is not natively supported by the hardware, it can still be supported at the software level by the operating system or the application framework, using for example time-sharing.

Such asynchronous communication is better served by a more abstract pattern based on asynchronous messages or events, modeling that something has happened that is potentially of interest to some other objects. Distributed infrastructures based on this paradigm are often called "event channel" or "software bus", based on the Observer pattern (or sometimes called Publisher/Subscriber pattern or Supplier/-Consumer pattern) in a distributed manner. The concept of such a software bus is quite similar to the Observer pattern: all consumers (i.e. observers or subcribers of the events) are connected to a shared medium called a "channel". They announce their interest in a certain topic (i.e. type of events) by subscribing to it. Objects or processes that want to send a message or an event to the consumers publish it under a certain topic to the bus. When that happens, and if the consumers's topic matches the supplier's topic, the message is forwarded to the consumer.

From a software design point of view, such a pattern offers the benefits of direct notification instead of busy-polling: an object or a component tells the environment to inform it when something happens, and when something happens it is then informed and it reacts accordingly. This pattern eliminates the need for consumers to periodically and constantly poll for new events. Furthermore, the concept easily allows the use of the Adapter pattern that some objects act as programmable middlemen in the event chain. These Adapter objects may multiplex, filter, or aggregate events and more importantly, possibly make the consumers and the suppliers "pluggable" into the system without any knowledge of their peers (Fowler, 2002).

This type of pattern is attractive for asynchronous communication in distributed systems due to its conceptual simplicity and has the benefit of decoupling objects in both space and time, while at the same time abstracting from the network topology and the heterogeneity of the components. However, it requires a powerful broker mechanism that takes care of event delivery, and requires the implicit or explicit semantics of the proxy mechanism to be carefully designed and analyzed – for example it is not clear how quickly events are to be delivered, whether events are to remain in the channel for a certain period of time so that subscribers who miss anything due to system failure can catch up on missed events, and whether there are any guarantees on the delivery order. It should be noted that the *Observer* pattern is basically a multicast scheme. It is well known in distributed systems theory that broadcast semantics, in particular for dynamic environments, is a non-trivial issue.

From this overview of communication patterns, it becomes clear that there is a trend in distributed systems: the communication paradigm is evolving from the simple procedural paradigm requiring relatively little system support, via the object-oriented method invocation principle, towards distributed infrastructures and more abstract schemes for loosely coupled asynchronous systems that require complex runtime infrastructures. Since most of the system models are evolving towards the asynchronous communication and processing paradigm, it is important to design the distributed multimedia system along with this trend so that the involved components, and the system and the environment can talk to each other in the same manner. Anyhow, asynchronous communication is indeed required for parallel and interactive media presentations in distributed settings. Scalability and efficient realization of such infrastructures is a real challenge, though. Next this challenge will be dealt with step by step, starting from the pattern of the event and data channels.

7.2 Channel

7.2.1 Intent

The Channel pattern decouples the communication between objects that are on the same host or distributed over a network, by defining two roles for the objects: The supplier role and the consumer role. Suppliers produce data sent to the channel and consumers process data received from the channel without directly interacting with each other, even when the other side is not available.

7.2.2 Context and forces

Actors are distributed active objects. They reside in multiple hosting devices and service the actions according to the requests from the clients (director or other actors). The most common communication pattern between clients and servers on a network is the client-server model as shown in figure 7.1.

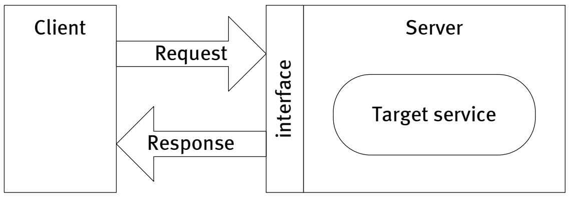


Figure 7.1: Client-server communication model

With this model, data is communicated by a client invoking operations on a service object on the server by sending a request. The request normally contains data arguments that are marshalled (packaged) and sent over the network to the service object. Once a client sends a request to a server, it is blocked and unable to perform any other function. The server side unmarshals the request to obtain the data, performs the necessary operations and returns a response back to the client. When the client has received the response it may continue its execution. For such communication to happen, there must be a client and a server available for the request call to succeed. If a request fails for any reason, for example the server is unavailable, clients will receive an exception and must take appropriate action. This is the common principle of all RPC or RMI based systems.

In many cases, this basic communication paradigm is sufficient and greatly eases development since it develops the distributed computing in a straightforward manner from normal non-distributed function calls. However, in some scenarios a more decoupled communication model between the clients and the servers is required. For example in our case of distributed multimedia:

1. The actors of a particular environment differ and their configurations may change during the presentation. The identities of the actors and their action services are unknown to the clients and therefore can not be connected directly.

2. The actors may move out the reach from the clients during the presentation, and the clients may wish to send requests to actors when they are not available. In this case, there needs to be a mechanism that buffers the requests until an actor becomes available again.

3. The clients, especially the director, have to be responsive to other events and can not afford to block while waiting for a response from one actor.

7.2.3 Solution

Techniques beyond the basic client-server communication paradigm are needed to overcome the problems just mentioned. One of the techniques is to add another layer between the client and server, for example the Common Object Request Broker Architecture (CORBA) Event Service (Bolton, 2001; OMG, 2004a), the Elvin notification service (Segall and Arnold, 1997), the InfoBus data controllers (Sun Microsystems, Inc., 1999) and the iBus channels (SoftWired AG, 1998). From design pattern points of view, these solutions are similar and fall into the same pattern: The clients are modeled as data consumers and the servers are data suppliers, and the data is communicated through middleware services, often referred to as channels. The pattern formalized here is based on the CORBA Event Service, with channel management and exception handling simplified or removed.

The Channel pattern provides a facility that decouples the communication between objects. It provides a supplier-channel-consumer communication model that allows multiple supplier objects to send data asynchronously to multiple consumer objects through a data channel. For example, the supplier-consumer communication model allows an object to communicate an important change in state, such as "an actor has just finished presenting a media object", to any other objects that might be interested in such an event.

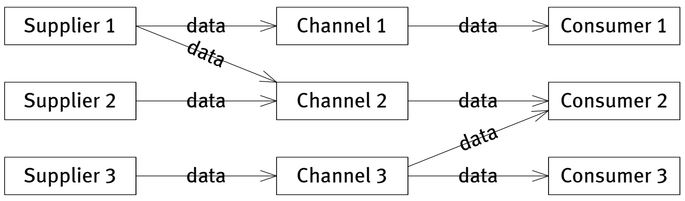


Figure 7.2: Supplier-Channel-Consumer communication model

Figure 7.2 shows three supplier objects communicating through three data channels with three consumer objects. Supplier 1 is connected to Channel 1 and Channel 2, and the data it supplied will be sent to Consumer 1 and Consumer 2. Consumer 2 is connected to Channel 2 and Channel 3 and it will receive data from all 3 suppliers. The channels are both consumers and suppliers. For a supplier the channel acts as a consumer of data at the supplier's site; for a consumer the channel acts as a supplier at the consumer's side. This is done through the use of proxy objects.

Instead of directly interacting with each other, suppliers and consumers obtain a proxy object from the channel and communicate with it. Supplier objects obtain a consumer proxy and consumer objects obtain a supplier proxy. The channel facilitates the data transfer between consumer and supplier proxy objects. Figure 7.3 on the facing page shows how one supplier (Supplier 3) can distribute data to multiple consumers (Consumer 2 and Consumer 3).

There are four general models of supplier-consumer collaboration in the Channel pattern. Figure 7.4 on page 84 shows the collaborations between suppliers, consumers and channels in each of the models outlined as follows:

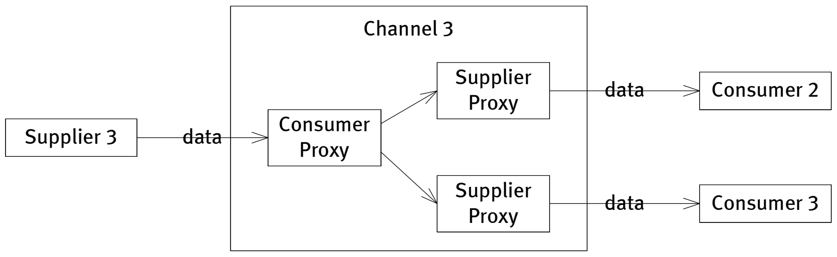


Figure 7.3: Consumer and supplier proxy objects

Canonical push model: In this model, event suppliers initiate the transfer of data to consumers. As shown in figure 7.4(a) on the following page, suppliers are the active initiators and consumers are the passive targets of the requests. Channels play the role of a *notifier*, as defined by the Observer pattern (see definition on page 57), where observers of an object are notified whenever the object changes it state. Active suppliers therefore use channels to push data to passive consumers that have registered with channels.

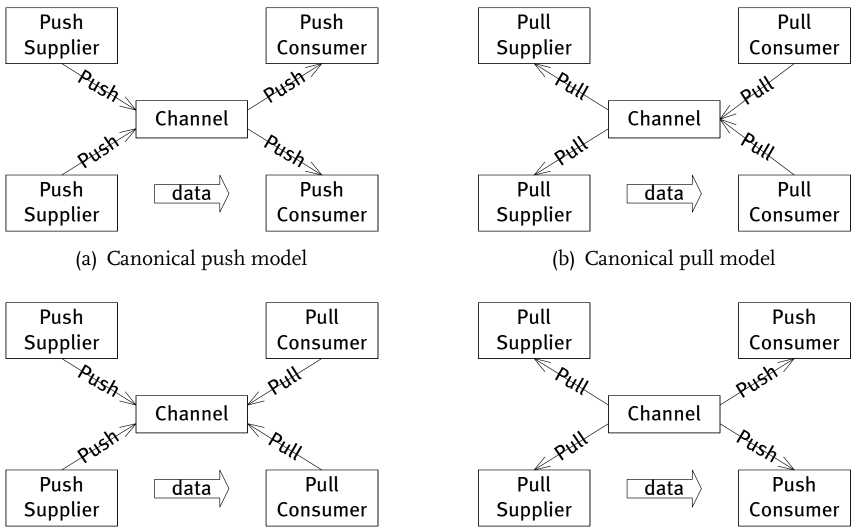
Canonical pull model: In this model, consumers request events from suppliers. As shown in figure 7.4(b) on the following page, consumers are the active initiators and suppliers are the passive targets of the pull requests. Channels play the role of a *procurer* since they obtain events on behalf of consumers. Active consumers therefor can pull data explicitly from passive suppliers via a channel.

Hybrid push/pull model: In this model, consumers request events that are queued at a channel by suppliers. As shown in figure 7.4(c) on the next page, both suppliers and consumers are the active initiators of the requests. Channels play the role of a *queue*. Active consumers therefore can pull data that is explicitly deposited by active suppliers via a channel.

Hybrid pull/push model: In this model, a channel pulls data from suppliers and pushes them to consumers. As shown in figure 7.4(d) on the following page, suppliers are passive targets of pull requests and consumers are the passive targets of pushes. Channels play the role of an active *agent*. These channels therefore can pull data from passive suppliers and push it to passive consumers.

7.2.4 Structure

Figure 7.5 on page 85 shows the static structure of the Channel pattern in a UML class diagram. A *Channel* object has two managers *SupplierAdmin* and *ConsumerAdmin* maintaining proxy consumers and proxy suppliers. A supplier can obtain a proxy consumer from the *SupplierAdmin* and connect to it. Either a *ProxyPushConsumer* object or a *ProxyPullConsumer* object can be used, depending on the data delivery model. At the another side, a consumer can obtain a proxy supplier from the *ConsumerAdmin* and connect to it. The consumer may decide which data delivery



(a) Canonical push model (b) Canonical pull model

(c) Hybrid push/pull model (d) Hybrid pull/push model

Figure 7.4: Data delivery models in Channel pattern

model to be used, by connecting to a *ProxyPushSupplier* object or a *ProxyPullSupplier* object, independently of the data delivery model used at the supplier side. See background material H for detailed Object-Z specifications.

Channel composition

The example above shows that the registration of a supplier or consumer is designed to be a two step process. A data-generating application first obtains a proxy consumer from a channel, then connects to the proxy consumer by providing it with a supplier. Similarly, a data-receiving application first obtains a proxy supplier from a channel, then connects to the proxy supplier by providing it with a consumer.

It might be unnecessary at the first sight to split such a process into two steps. The advantage of the two-step registration process over the single step one is the possibility of composing channels by an external object or process. With the two step process, it is possible to compose two channels by obtaining a proxy supplier from one and a proxy consumer from the other, and passing each of them a reference to the other as part of their connect operation (see figure 7.6(a) on page 86).

It is possible to chain the channels to create a data filtering graph that the consumers use to receive a subset of the total events from the suppliers. In figure 7.6(b) on page 86, Channel B and C as consumers of Channel A, can for instance select a subset of the received data to pass on to their own consumers. Hence Consumer A and B will only receive the data selected by Channel B, and Consumer C and D will only receive the data selected by Channel C. Instead of letting Consumers to pick up interested data by themselves, the intermediate channels centralize the

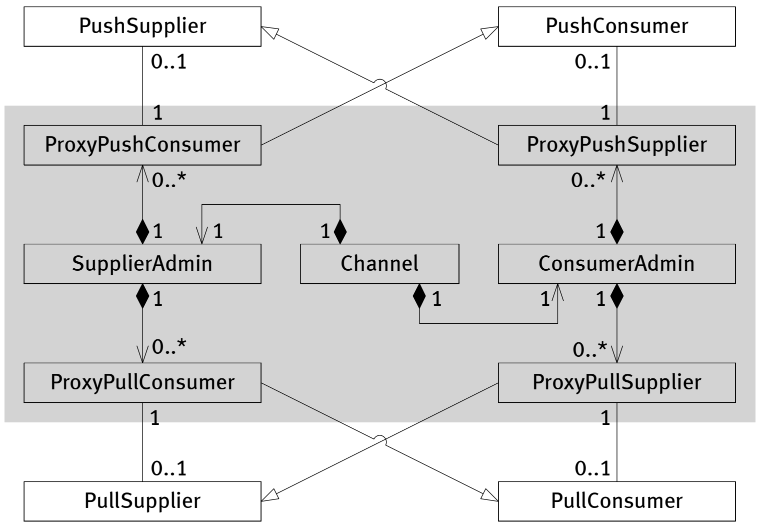


Figure 7.5: Channel pattern

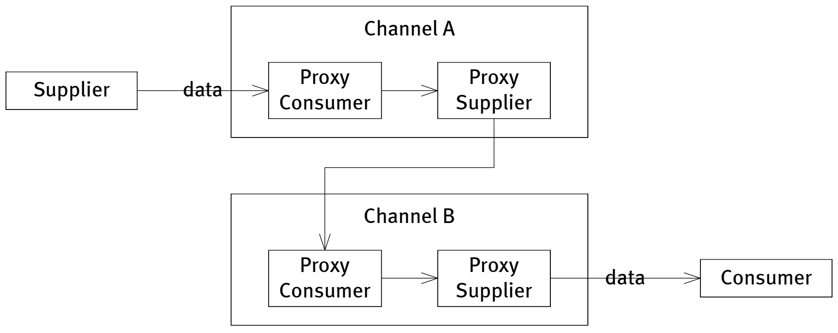
process for the consumers that are interested in the same subset of data. This somehow compensates the overhead of the data broadcasting.

This is useful in the IPML system when their is a need to reduce the load of the event and message propagation in the distributed network, but not to change the controlling hierarchy. A connected actor may disconnect from its channel to the director or to another actor, apply a new channel with new event and message requirements, then connect itself to the new channel and in turn the new channel to the original channel. New filtering requirements are then enforced, but from the viewpoint of the director or the others, nothing has been changed.

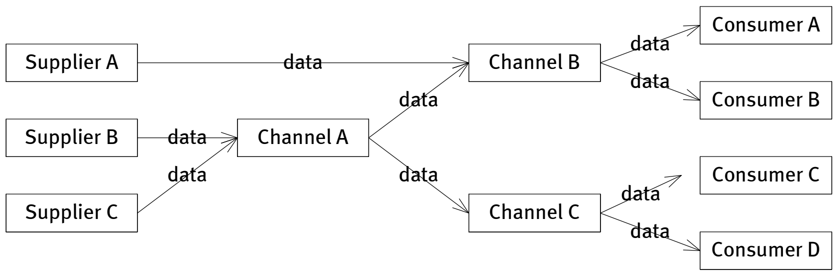
7.2.5 Consequences

Benefits The Channel pattern provides the following benefits:

Decentralized communication: The channels work in a distributed environment. The design does not depend on any global, critical, or centralized service, as long as the supplier and the consumer have access to a shared channel. The suppliers can generate data without knowing the identities of the consumers, and the consumers can receive data from the channel without knowing the generating suppliers. In the IPML system, by applying the channel pattern, the director does not take the central position of managing the communication. The actors can connect themselves to the channel services even when the director is not active or not available. This is essential in a dynamic configuration.



(a) Two step registration



(b) Data filtering through channel composition

Figure 7.6: Channel composition

Asynchronous communication: If clients and servers are connected through channels, the channels can play the role of a queue and the clients do not have to block to make requests. This makes the client applications much easier since it does not have to be multi-threaded. Clients and servers can also run at different times. For requests to be completed successfully, both the client and the server do not have to be available at the same time. If a server has failed and a client sends data to the failed server, the data remains in the queue until the server becomes available when it can retrieve the process the message accordingly. For our distributed multi-media system, it also supports nomadic actors. Disconnected from the director, actors will be able to accumulate outgoing requests in queues until the connection with the director is established, and vice versa.

Flexible communication: The Channel pattern allows multiple consumers and multiple suppliers. A supplier can issue data to all consumers at once. The consumers can either request data or be notified of the data, whichever is more appropriate for the design and the implementation. The pattern is also capable of being implemented and used in different operating environments, for example, in environments that support multi-threading and those that do not.

Liabilities However the Channel pattern also has the following liabilities:

Over generalization: The Channel pattern is very flexible and can support a wide range of applications. As a consequence the registration of suppliers and consumers is rather complicated due to the multi-step connection process to the channel.

Limited data filtering: The Channel pattern defines Channels as broadcasters that forward all events from suppliers to all consumers. This approach has several drawbacks. If consumers are only interested in a subset of types of data from the suppliers, they must implement their own data filtering to discard unneeded data. Furthermore, if a consumer ultimately discards data, then delivering data to the consumer needlessly wastes bandwidth and processing. Data filtering is possible by chaining the channels together to create a data filtering graph as showed in figure 7.6(b) on the facing page. However the filter graph increase the number of nodes that data must travel between suppliers and consumers and hence the overall cost.

No data correlation: A consumer may only be interested in the data from particular suppliers, and may wish to specify more complex conjunctive (AND) or disjunctive (OR) semantics when specifying filtering requirements, based on the type or the supplier of the data. With such correlation mechanisms, the bandwidth and processing load can be greatly decreased when there are many consumers and providers are communicating through the channels. In the IPML system, if the communication among a big number of actors needs to be managed, the efficiency of event and message propagation is important and a more effective filtering mechanism is needed.

No Quality of Service (QoS) Support: The Channel pattern does not specify the Quality of Service (QoS) that the channels must implement. This means different implementations may have different level of QoS. This means the reliability, availability, throughput, performance and scalability requirements have to be taken care with extensions. Clearly no single implementation of the channel pattern can optimize such diverse technical requirements. QoS support is especially important for time critical scheduling tasks, and performance demanding multimedia data transport in the IPML system.

Confirmation of reception Some application may require the consumers provide an explicit confirmation of reception back to the supplier. This can be done effectively using a “reverse” channel through which the confirmation can be sent back as normal data. However, strict atomic delivery between all suppliers and all consumers requires additional interfaces.

7.3 Real-time Channel

7.3.1 Intent

The Real-time Channel pattern improves the Channel pattern to provide a flexible model for asynchronous communication among distributed components in real-time applications, with efficient data filtering and correlation, predictable and scalable data dispatching, and Quality of Service (QoS) support.

7.3.2 Context and forces

Distributed multimedia applications, like other distributed real-time applications, require stringent real-time QoS support, although the requirement is not as stringent as the systems like sophisticated medical operation and space mission computers where failure to meet deadlines can result in loss of life or significant loss of property. Failure to meet the timing, priority and reliability requirements in distributed systems man cause desynchronization, latency and jitter in content delivery, hence an unpleasant media experience for the end users. For real-time distributed multimedia, the following issues are important to be covered:

Reliability: The Channel pattern described in previous section handles asynchronous communication in many flexible and versatile ways, to decouple the supplier and the consumer and increase concurrency and prevent unnecessary blocking. However, multimedia applications require the communication to be not only asynchronous, but also reliable. Some data, for example certain user interaction events, must delivered no matter how much it is delayed, even when the source or the target device is temporally disconnected from the system. Other data, for example a sound effect that is designed for a particular moment, should be dropped if it missed its deadline or after the system has failed to deliver on is best effort. The channel should provide options for both reliable persistent delivery and best effort delivery.

Timing: Distributed multimedia applications require not only high transmission reliability, but also stringent delivery delay. In a distributed environment, delay-free communication is not possible, but for multimedia applications, the delay must be controlled to keep the minimum synchronization, otherwise the intended temporal relationships within and among multimedia objects would be distorted due to the delay jitter(Liu and El Zarki, 1999). The Channel pattern should be extended to allow the data supplier or the consumer to specify delivery time constraints according to their needs.

Priority: A *Channel* is an objected shared by multiple suppliers and consumers, and data may be queued for concurrent delivery, including scheduling commands, user interaction events and chunks of multimedia content. Scheduling commands, for example “Stop” applying to a media stream being transmitted in the same channel, should have higher priority than the media stream otherwise the stream would stop until the transmission is completed. The *Channel* object must support the preemption of the lower priority data so that the higher priority data can be delivered first.

Data filtering and correlation: Data filtering and correlation should be supported for the multimedia applications. Although the large amount of data transmission of multimedia data can be handled with dedicated streaming protocols and channels, smaller media data chunks can be transmitted among distributed components for interactive applications. The broadcasting scheme of raw data transmission in the Channel pattern may cause high load on network and processing resources when the system scales up with many supplier and consumer nodes. To ensure the scalability, the Channel pattern should be improved with data filtering and correlation mechanisms.

7.3.3 Solution

Many real-time applications, such as industrial process control and military command/control systems, commonly use the push model because of the efficient and predictable execution of operations (O'Ryan et al., 2001; Rajkumar, Gagliardi, and Sha, 1995). The consumers only react when their dependencies are satisfied, that is, when the data is pushed from the supplier proxies. In contrast, the pull model requires the consumers to actively acquire data from the supplier proxies. The consumer process would need to be blocked until there is data to be pulled. In order to do this, the system must allocate additional threads to allow the application to proceed while the pulling process is blocked. Adding extra threads may increase context switch overhead and synchronization complexity; it may also require complex real-time scheduling polices. These negative consequences made the pull model often not appropriate in real-time tasks. Therefore, for real-time scheduling and synchronization tasks in our distributed multimedia system, let's focus on the push model.

The solution is to extend the push model of the Channel pattern with extra components and interfaces, so that the consumers may subscribe their interests (interested subset of data, source of the data and QoS requirements) to the channel. The channel includes a dedicated component to filter the incoming data for the consumers according to their subscription. The channel then enqueues the data according to the priority and the displacing order. At the same time, the channel employs an separate active process that keeps dequeuing data from the queue and dispatching it to the connected consumers.

7.3.4 Structure

The Real-time Channel pattern is an extension of the push model of the Channel Pattern. Figure 7.7 shows overall the structure of the extension. The *RTChannel* contains additional components to manage the details of the communication:

Filter when the data arrives from the *ProxyPushConsumer*, the *RTChannel* forwards it to its *Filter* component to determine which *RTProxyPushsupplier* objects (in turn, the connected *PushConsumer* objects) should receive the data and when to dispatch the data.

Scheduler The filtered data, together with the interested proxies, are then forwarded to the *Scheduler* to calculate the preemption priority, and within the same priority, the dispatching order.

PriorityQueue For each supported preemption priority, the *RTChannel* maintains an ordered queue. The data and the targeted proxy are inserted with corresponding preemption priority. The queue maintains its elements in the dispatching order.

Dispatcher The dispatcher is responsible for removing data/proxy tuples from the priority queues and forwarding the data to the targeted proxy by calling the *Push* operation. Depending on the placement of each tuple in the priority queues, the Dispatcher may preempt a running *DispatchingThread* with the preemption priority to dispatch the data.

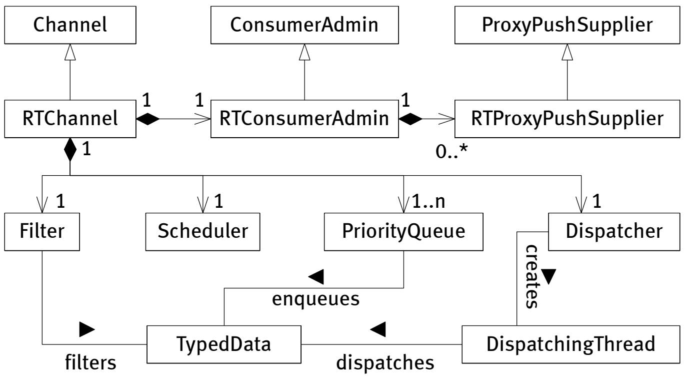


Figure 7.7: Real-time Channel

The classes *RTConsumerAdmin* and *RTProxyPushSupplier* extend *ConsumerAdmin* and *ProxyPushSupplier* respectively to accommodate the subscription and unsubscription operations for the consumers to register their interests to the *Filter*. The interests are modeled as *DataInterest*, which contains the filtering conditions on the source of the data, the type of the data and the QoS requirements. The Real-Time Channel pattern requires that Any data to be transferred through real-time control facilities must implement the *TypedData* attributes, or the data will be transferred as it is in the Channel pattern. The QoS requirements are modeled as *QoSProperties*, a list of *name* $\mapsto$ *value* pairs.

Figure 7.8 on the next page shows roughly the data flow inside the *RTChannel*. See background material I for detailed Object-Z specifications.

7.3.5 Consequences

Since the Real-time Channel pattern is an extension of the Channel pattern, it provides all the benefits that the Channel pattern does. In addition, with special attention paid to the QoS needs of distributed real-time applications, the Real-time Channel pattern also provides the following benefits:

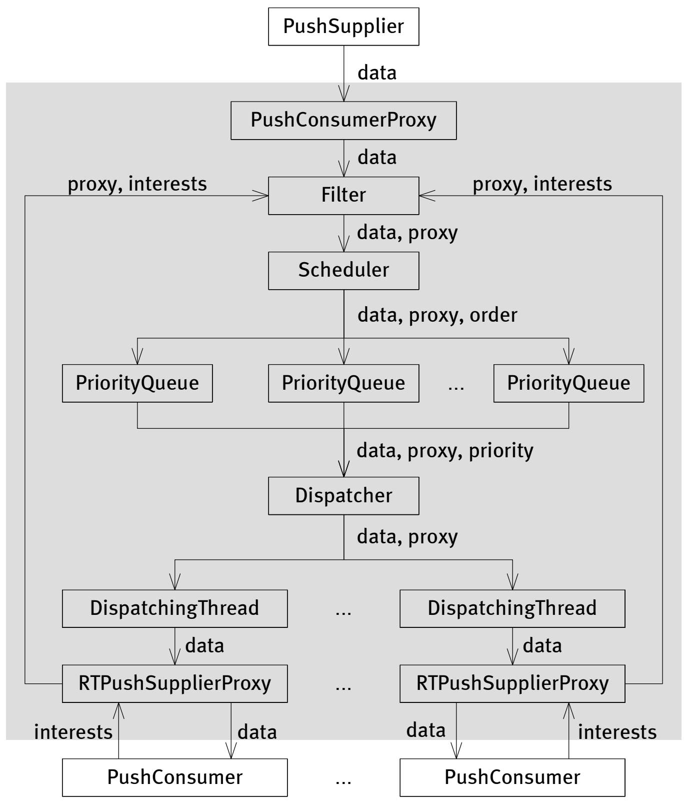


Figure 7.8: Data flow in a Real-time Channel

Benefits

Flexible QoS control: The QoS requirements are modeled as a list of *name* $\mapsto$ *value* property pairs. Although several QoS properties are defined as an example, the structure of the pattern does not depend on the elements of the QoS list. If nothing is specified as QoS requirement, the Real-time Channel still works as a signal queue, multi-thread dispatching channel. If a new QoS property is introduced, this structure of separating filtering, scheduling and dispatching can easily incorporate the new QoS property with limited change to the related components. For example, if a more detailed timing requirements on dispatching would be introduced, a QoS property *EndTime* might be introduced to require that a transaction must be finished before a certain time, in addition to the property *Deadline* which specifies the deadline of the staring time. The scheduling and dispatching strategies based on this *EndTime* requirement can be easily implemented by extending the *PriorityQueue*, the *Scheduler* and the *DispatchingThread* with *EndTime* being taken into account, without touching the basic structure and changing the interfaces.

Flexible scheduling and dispatching policies: The *Scheduler* and the *Dispatcher* in this pattern can be replaced or extended to introduce different scheduling and dispatching policies. It is also possible to equip the channel with multiple scheduling and dispatching policies thus the configuration of the scheduling and dispatching strategy can be done at the runtime. Further, because of the loose coupling of scheduling and dispatching components from the other components, it is possible to employ the Plugin pattern (Fowler, 2002) so that the *Scheduler* and the *Dispatcher* with different policies can be installed and removed on the fly during the runtime.

Multi-thread dispatching and predictability: The Real-time Channel pattern uses multiple threads within the channel forwarding data to the consumers. It can be configured with an application-specified strategy to assign the number and priority of real-time threads that will dispatch the data. The specification here defines example strategies to show how the application-specified needs can be covered. A dispatching task can be assigned to a thread at the appropriate priority, avoiding priority inversions in the channel and thus achieving better predictability by enforcing scheduling decisions at runtime.

Event filtering and correlation: Although it is possible to broadcast the data to connected consumers and let the consumers perform its own data filtering, it wastes network and processing resources and requires extra work from developers. The channel in this pattern sends data to a particular consumer only if the consumer has subscribed for it explicitly, or has not subscribed for any data. This pattern allows not only disjunctive dependencies by uniting multiple *DataInterest* objects, but also conjunctive dependencies by the consumer defining conditions for the source, the type and the QoS requirements of the data.

Liabilities However the Real-time Channel pattern also has the following liabilities:

Data marshalling and unmarshalling: The QoS and filtering features of this pattern requires the data to be typed. The supplier must first transform it into a ↓*TypedData* object with the source and type information and QoS requirement, marshal the object into a data stream , then push the data stream it into the channel; the channel must analyze the type and the QoS requirements by partially unmarshalling the stream; every consumer must unmarshal the stream in order to peel out the data transmitted. This procedure would not take too much of the processing resources if the task is to transmit a small chunk of messages or events. However, if the data itself is a continuous stream with a complicated structure, for example, when it is needed to transmit a stream of video frames, this pattern does not apply. With this pattern, the transmission can only happen when the whole package of the data is marshalled into the required format, which does not cover the concept and the needs of continues streaming media.

Less generality: While Channel patten might have a liability to be too general with many possible combinations of the push and pull models, the Real-time Channel pattern has included details of filtering, scheduling and dispatching, with focuses on the QoS and real-time requirements. These requirements have to be taken care of everywhere for real-time scheduling and event dispatching, however, as a pattern it has less generality than the Channel pattern.

Bandwidth and throughput not handled: While the QoS requirements on timing, priority and reliability are covered with examples in the specification here, other requirements such as those on channel bandwidth and periodic communication throughput have not been taken into account. To handle these bandwidth and throughput requirements, the Real-time Channel must be considered in a bigger context: channels may share the same physical connection; the number of the sharing channels and the total bandwidth of the physical connection would influence the periodic throughput. From the architecture point of view, these requirements are no more than some additional *QoSProperties* to be handled by the *Scheduler* and the *Dispatcher*. However, these requirements require more sophisticated bandwidth resource scheduling in addition to task scheduling, which is considered to be too detailed for a pattern level specification. As an interesting topic, let's leave it as an option for future research.

Please refer to background material I for the details mentioned in the above discussions.

7.4 Streaming Channel

7.4.1 Intent

Extend the Channel pattern to support multimedia streaming data, quality of service control, as well as multicast data delivery.

7.4.2 Context and forces

The use of multimedia data requires more effective and faster communication. This requirement places new demands on on the enabling technologies and the middleware that creates distributed applications. Support for multimedia in the Channel pattern, and the Real-time extension as well, do not have specific support or facilities for handling media streams. The patterns specified in the previous sections assume that the events or messages transferred between distributed components contain relatively small discrete data entities, which are sufficient to notify some change of application state or send controlling commands, and do not require much of processing resources and network bandwidth. However this assumption does not hold any more when multimedia applications need to use and transport special forms of data, such as audio and video streams, or messages and events that contain multimedia streams. These streams are continuous, processing intensive, and require stable communication bandwidth to bring the data across. Frequent marshalling and unmarshalling of the transferred objects during the communication have been proven inefficient for this purpose (Gokhale and Schmidt, 1998).

There are many technologies and protocols available for streaming multimedia over a network. While Real Media, Apple QuickTime and Microsoft Media support proprietary streaming media types and technologies, many open standards such as MPEG-2 and MPEG-4 are also widely supported. Multimedia can be streamed and controlled over networks using protocols such as Real-time Transport Protocol (RTP), Real Time Streaming Protocol (RTSP), Real-time Transport Control Protocol (RTCP), and Microsoft Media Services (MMS). Most of these protocols are built on top of User Datagram Protocol (UDP), but may also work on top of more reliable protocols such as Transmission Control Protocol (TCP).

It seems that these technologies and protocols would be enough for streaming data between components. However they imply a tight coupling between the client (consumer) and the server (supplier) and it's often difficult to separate the media distribution protocols from the rendering technology being used. Further, the multimedia data may need to be exchanged among loosely coupled and distributed components in response to application state changes. A many to many, more flexible and decentralized communication structure is needed.

7.4.3 Solution

Since there are many dedicated protocols and technologies available for streaming multimedia data with quality of service support, let's leave the data streaming task to these dedicated protocols and technologies. These streaming connections are modeled as objects of the type *Stream*. A streaming channel manages multiple *Streams*, each of which is created with an identity of type *String* and with QoS requirements. Given the identity of a stream, *StreamSuppliers* and *StreamConsumers* may connect to the stream in the channel by connecting to a proxy consumer or a proxy supplier. A *StreamSupplier* pushes data into the channel, the connected *ProxyStreamConsumer* pushes the data into the *Stream* and the *Stream* does the streaming according to the preset QoS requirements. At the another side, an Observer pattern is applied – *ProxyStreamSuppliers* that are related to this stream will be notified when the data arrives. Each of the related *ProxyStreamSuppliers* will then push the data to the connected *StreamConsumer*. The *StreamSuppliers* and *StreamConsumers* do not directly connect to each other but through the streams that are managed by the channel. They may disconnect themselves from the channel without affecting other components communicating through the same or a different stream.

The structure of this solution is very similar to the push model of the Channel pattern, in the sense that the data is pushed into the stream and the stream pushes data to the consumers. The difference is that the channel here acts as a stream manager and it does not directly serve data transport. Since the data transportation is rather different, it is not wise to extend the push model and override the semantics. Instead, the Streaming Channel pattern extends the Channel pattern with a new model of data streaming, keeping the existing pull and push models intact.

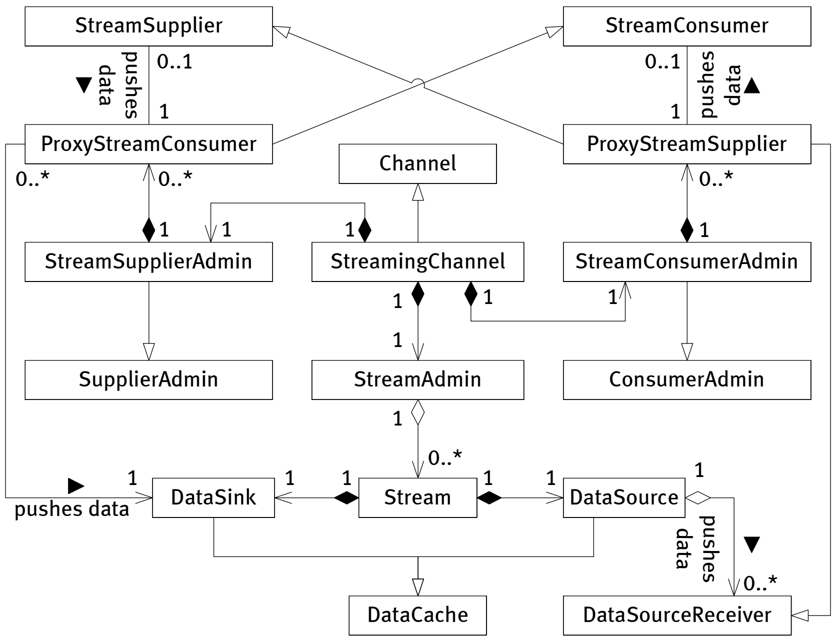


Figure 7.9: Streaming Channel pattern

7.4.4 Structure

Figure 7.9 shows the static structure of the Channel pattern in a UML class diagram. A *StreamingChannel* object has two managers *StreamingSupplierAdmin* and *StreamingConsumerAdmin* extending *SupplierAdmin* and *ConsumerAdmin* respectively, maintaining proxy consumers and proxy suppliers. It also has a *StreamAdmin* maintaining *Stream* objects. A supplier can obtain a proxy consumer from the *StreamingSupplierAdmin* and connect to it as it does in the Channel pattern. In addition to *ProxyPushConsumer* objects and *ProxyPullConsumer* objects, *ProxyStreamConsumer* may also be used if the required *Stream* has been created. At another side, a consumer can obtain a proxy supplier from the *StreamingConsumerAdmin* and connect to it. The consumer may decide which data delivery model to be used, by connecting to a *ProxyPushSupplier* object, a *ProxyPullSupplier* object, or if the consumer knows the identity of the *Stream*, a *ProxyStreamSupplier* object. The push and pull models of the Channel pattern are fully inherited by this pattern, though the details of these two models are not shown in this figure. See background material J for detailed Object-Z specifications.

7.4.5 Consequences

For certain details mentioned below, please refer to background material J.

Benefits The Streaming Channel pattern extends the Channel pattern with a streaming data transportation model, without infecting the original push and pull models. Hence it provides all the benefits that the Channel pattern does. With the streaming style, this pattern also provides the flowing benefits in addition:

Easy to set up streams: Using stream-type channels makes it straightforward to set up and manage multimedia data streams. This can be done without having to manually allocate multicast addresses or port numbers and then design a suitable multicast protocol at the application level.

Support for multimedia events: Multimedia data can be exchanged in the context of asynchronous event notification, which makes this pattern widely applicable for applications that require notifications of events with multimedia content.

QoS support: The channel in this pattern acts as a stream manager and leaves the data transportation to dedicated streaming technologies and protocols, which ensures efficient delivery of data to consumers based on the QoS parameters associated with each stream during stream creation.

Media type Independency: Support for streaming data in this pattern is provided in a manner that is independent of the type of the multimedia data being propagated and the technology used to capture and render the stream. This pattern does not place any requirements on multimedia types that may be used. It does not attempt to decode and interpret the data or the data fragment as it passes through the relevant channels.

Liabilities However the Streaming Channel pattern also has the following liabilities:

Limited Reliability: This pattern does not guarantee the data delivery. The data might be dropped before, during and even after it is streamed, because of either a small cache or a limited transport rate.

The reliability of the data transportation also depends on the underlying streaming protocol. Most of the real-time streaming protocols are build upon UDP. UDP sends the media stream as a series of small packets, called datagrams. This is simple and efficient; however, packets are liable to be lost or corrupted in transit. Depending on the protocol and the extent of the loss, the client may be able to recover the data with error correction techniques, may interpolate over the missing data, or may suffer a dropout. Hence when UDP is used, it is not suitable for transferring scheduling commands and user interaction events. A more reliable pattern, for example the Real-time Channel pattern should be applied. Or, the channel should create reliable streams by using a more reliable protocol, such as TCP. It guarantees correct delivery of each bit in the media stream. However, it accomplishes this with a system of timeouts and retries, which makes them more complex to implement. It also means that when there is data loss on the network, the media stream stalls while the protocol handlers detect the loss and retransmit the missing data.

Stream id negotiation: Only the suppliers can create streams with a specified *id*. For both suppliers and consumers to communicate with same stream, they must agree on the *id*. Either this *id* is predefined and shared as a protocol, or the *id* must be negotiated. The negotiation requires a different connection other than the streaming model. Since in a distributed multimedia system, the streaming model is not reliable enough to transmit controlling commands and interaction events, reliable push or pull models are needed anyway. The negotiation can then be done through these reliable channels. For example, a supplier may create a real-time stream with an *id* "rtsp://supplier.com/live/video" in a Streaming Channel, and push this *id* over the Real-time Channel to the consumer. Then both the supplier and the consumer may connect to this stream with the same *id* for streaming the live video.

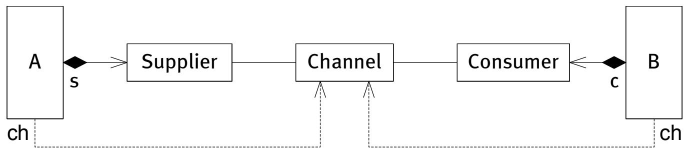
Multicast streaming: Connecting multiple consumers that are distributed over multiple systems to one stream require the underlying technologies and protocols support efficient multicast data delivery. Otherwise, if only unicast is supported, a supplier must create multiple streams for the consumers that wish to receive the data. Multicasting delivery is then possible, however the supplier must maintain a list of streams and iterate through this list to deliver the data.

Compatibility The streaming model is compatible with neither the push model nor the pull model. Unlike the push and pull model that can be combined with each other to form different communication styles, the streaming model can not be mixed. If the streaming model is used, it must be used by both the suppliers and the consumers. Data sent through streaming will not received by any push or pull consumer but only the streaming consumers; and data sent using either push or pull model will not arrive at any streaming consumer.

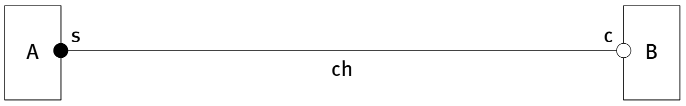
7.5 Concluding remarks

The channel is an abstract concept, which is implemented as a software service that runs across the distributed components. Different protocols and hardware connections can be used for data transportation, which would have incurred many complex and trivial issues that should not appear at an architectural level, if the concept of the channel is not introduced. Separating the suppliers and the consumers with channels enables decentralized, asynchronous and flexible communication, and issues such as QoS control and data filtering can be taken care of by adjusting the channel service.

The patterns described in this chapter fall into a general pattern: no matter whether they use the push model, the pull model or the streaming model, no matter whether they include the mechanisms of data filtering and QoS control, they have three types of basic elements in common: data suppliers, data consumers and a



(a) UML diagram



(b) Simplified diagram

Figure 7.10: Simplified view of channels

channel in between. At a more general level, an object that needs to communicate asynchronously with other components may aggregate or include suppliers or consumers as its input or output components. In the UML diagram shown in figure 7.10(a), the class *A* has a (instead if "is a") supplier *s* and the class *B* has a consumer *c*. Both *A* and *B* have access to a global *Channel* object *ch* and connect their *s* and *c* components through *ch*.

The connection pattern shown in figure 7.10(a) is often used in our distributed system. To simplify the diagrams and the specifications, an equivalent diagram will be used, as shown in figure 7.10(b), where a "●" indicates a supplier component, a "○" indicates a consumer component and a connecting line in between indicates the channel. The process of obtaining and connecting to a proxy consumer or a proxy supplier from a channel will also be omitted hereafter from the specification, if a connection line between the supplier and the consumer is shown in the diagram.

These components can be considered as output ports, input ports and channels connecting these ports. The requirements on the communication can be modeled as the properties or functions of the channels. When the details of the communication are not any more the main focus, this approach would provide a more abstract view on the higher level architecture about the interfaces, the distribution and the state transitions. For example in Broy's component-based logic for formalizing distributed systems, components are specified by logical expressions relating input and output histories of communications on their channels, described using state machines (Broy, 1999). Here it shall not be attempted to analyze formally in which sense that the object-oriented view on the communication is equivalent to Broy's component-based view. These two views show clear similarities, for example objects in Object-Z and components in Broy's theory are all state machines, input and output ports are all connected through channels etc. While the Object-Z approach is closer to object-oriented implementation, Broy's theory gives a more abstract and more manageable view on the component-based communication and architecture. Later, this approach will be used for analyzing the mapping problems.

Having described the actions and the communication channels, now it is ready to present the actors – those who perform and communicate.

CHAPTER 8

Actors

Having defined the actions that actors perform to present the multimedia objects and the channels that these actors use to communicate with each other, it is now ready to examine the structure of these actors. The actors are in the system not only to present the multimedia objects, but also to provide the interface for the users to interact with the system. This chapter will first look at the interactive system architecture in general, then formally examine and compare two agent-based architectural models (MVC and PAC) in detail. PAC is selected as the overall system architecture, and the actors are implemented as PAC agents that are managed by the scheduling and mapping agents in a PAC hierarchy, connected with the channels defined in the previous chapter, and performing the actions defined in chapter 6.

8.1 Interactive system architecture

An interactive system architecture is a set of structures, including components, the outside visible properties of these components and the relations between them (Coutaz, 1997). Several architectural patterns have been proposed by researchers and practiced in software industry over the last twenty years (Bass et al., 1991; Buschmann et al., 1996; Coutaz, 1997; Pfaff, 1985; Stephanidis, 2001). These results can be grouped into three categories:

The language model: Foley and Dam (1982) propose to decompose the application and the user interface in the same way as a language. They suggest that each interaction with the system can be divided into four levels: 1. *semantic*: the meaning or purpose of the task; 2. *syntactic*: the structure or sequence of actions to complete the task; 3. *lexical*: the symbols used (e.g. words or icons) for each action; 4. *device*: the input device used to supply lexical components. The drawbacks of this model are that the priority is given to the shape of the information rather than the interaction itself and that the level of abstraction of the communication protocols is not sufficiently clear (Coutaz, 1987; Hartson and Hix, 1989; Knight and Brilliant, 1997). Despite these drawbacks, the basic principles of this model have been inspiring and have influenced the design of other generic models, for example, the functional models.

Functional models: These models provide a layered separation between the function core and the user interface. The Seeheim model (Pfaff, 1985) represents one of the first and "best" known models for user interfaces of this type (figure 8.1). It describes the user interface as consisting of the three components: 1. *presentation*, which performs a lexical analysis in order to translate the input of the user into into internal representation; 2. *dialogue controller*, which provides a syntactic analysis on the internal representation and manages the dialogues between the presentation and the application interface. 3. *application interface*, which translates the internal representation into function specifications in order to reach the application's function core. The Seeheim model showed to be inadequate for complex graphical user interfaces, thus being adapted, extended, or revised by other prescriptive models (Coutaz, 1987; Duce et al., 1991; Krasner and Pope, 1988; Lantz et al., 1987).

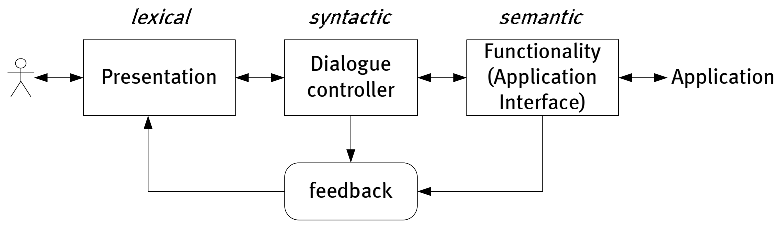


Figure 8.1: Seeheim Model

The Arch model or the Slinky metamodel (Bass et al., 1991, 1992) is one of these revisions. Instead of examining the functionality of an interactive system in order to separate the user interface from core functions, this model examines the nature of the data that is communicated between the user interface and the other components of an interactive system (figure 8.2 on the next page). The components *presentation* and *interaction* decompose the *presentation* of the Seeheim model. The presentation component supplies a set of presentation objects that are independent of any interaction toolkit and thus the interaction objects. The *domain adaption* component is used for adjusting the differences in the domain objects between the *functional core* and the *dialog controller*. Note that the term *object* is used as an expository abstraction for describing the transmitted information between the components. The term and the shape "Arch" suggest "the more stable development environment that occurs when goals have been set and choices have been made" (Bass et al., 1992).

Seeheim and Arch models set the foundations for functional partitioning. Modifiability, motivated by the iterative design of user interfaces, was the driving principle. The slogan was "*Separate the functional core from the user interface*". Seeheim and Arch provide canonical functional structures with

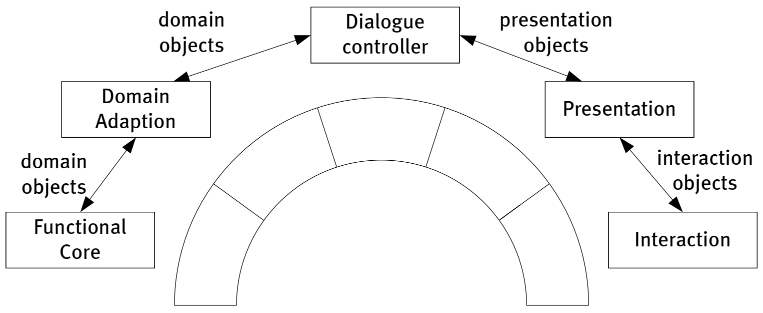


Figure 8.2: Arch Model

big conceptual blocks, which is useful for rough analysis and design of the functional decomposition of an interactive system. Multi-agent models aim at a more fine-grained and object-oriented decomposition.

Agent based models: Agent-based models structure an interactive system as an organization of specialized computational units called agents. This thesis will not detail the notion of agent, but retain the following definition: An agent can be a physical or virtual entity that can act, perceive its environment (in a partial way) and communicate with others, is autonomous and has skills to achieve its goals and tendencies (Ferber, 1999). Agents that communicate directly with the user are sometimes called interactors (Hu and Feijs, 2003a,b; Markopoulos, Johnson, and Rowson, 1998; Markopoulos, Shrubsole, and de Vet, 1999).

Agent-based models suggest grouping the three functional aspects (presentation, dialog control and functional core) together into one unit, the agent, according to the principles of object-oriented composition and encapsulation. The agents are then organized in a structure using event-driven mechanisms or communication channels. All of the agent-based styles and tools push forward the functional separation of concerns advocated by Seeheim and Arch. They generalize the distinction between the functional core and the user interface by applying the separation at every level of abstraction and refinement.

A number of agent-based models and tools have been developed along these lines, among which MVC and PAC models are typical agent-based styles or patterns. These models stress a highly parallel modular organization and distribute the state of the interaction among a collection of cooperating units. For applications that involve multiple units or devices, such as the IPML system, modularity, parallelism and distribution of these agent-based models are useful and convenient for designing user interfaces.

Chapter 3 has briefly introduced these two major interface architectures Model-View-Controller (MVC) and Presentation-Abstraction-Control (PAC). In the design of the StoryML system, PAC was found to be more suitable mainly because it was designed for distributed use. This decision was kept for the final design of the IPML system. In this chapter, a formal study on both of them is conducted and provides a more detailed insight into the reasons why PAC is more suitable for distributed situations, to consolidate the observations and the design decision.

Both of them have been studied formally by Hussey (1999); Hussey and Carrington (1997, 1996) using an Object-Z approach. A composite clock system was described as an example to compare MVC and PAC, especially on the presentation concerns, that is, how input is obtained and how output is produced. The communication between components is handled by aggregating the components and combining their operations in the aggregated component. Nowadays, MVC and PAC like structures are also used for distributed applications such as Web services, for example in Microsoft .NET (Microsoft, 2003) and Apache Struts (Holmes, 2004), where the MVC or PAC agents, and possibly their components, are distributed over the network. Direct access to other component's state and operations is not feasible. The communication between these components and agents has then to be done through network connections instead of direct functional calls or operation compositions.

Other agent-based models are also studied using different formal approaches. Abowd (1991) uses uses a hybrid of Communicating Sequential Processes (CSP) (Hoare, 1985) and the Z notation (Lightfoot, 2001; Spivey, 1992; Woodcock and Davies, 1996) to illuminate agent-based separation and usability properties of interactive systems. Markopoulos (1997) uses Language Of Temporal Ordering Specification (LOTOS) (Bolognesi and Brinksma, 1987) to describe his Abstraction-Display-Controller (ADC) interactors (Markopoulos et al., 1998). LOTOS is also used by (Stirewalt and Rugaber, 2000) to stress the composition issues of the interactive agents. Alexander (1990) uses CSP to decompose complex dialogs into separately defined application and presentation agents, while van Schooten (1999) uses CSP to model the interactive components in virtual environments. One may indeed gain many insights about the compositional structures and the interactive behaviors of these agents, but might still miss the parts about how these agents would cooperate and communicate in a distributed setting.

MVC and PAC are going to be compared using Object-Z again, but with more attention to the communication in distributed environments: distributed components and agents will be loosely connected through communication channels, using the Channel patterns from the previous chapter. By doing so, which of them fits better in a distributed system can be decided.

8.2 Distributed Model-View-Controller

The MVC pattern was first implemented in the Smalltalk-80 system as a user interface paradigm (Krasner and Pope, 1988). It has been gradually evolved into an user interface pattern for GUI (for example, JFC/Swing in Java (Walrath et al., 2004)) and architectural patterns for interactive systems (for example, Document-View in Microsoft Foundation Classes (MFC) (Kruglinski, 1995) and web applications (Holmes, 2004)). Each of them differs in either the responsibility of the controller, or the communication between the components. The established and widely accepted descriptions by Howard (1995) and Buschmann et al. (1996) are followed here.

8.2.1 Structure

In MVC, an interactive agent is divided into three components that are respectively responsible for *processing, output,* and *input*: The *Model* is about the domain-specific representation of the information on which the application operates. Domain logic adds meaning to raw data. The Model contains the core functionality and data that are independent of specific output representations and input behavior. The *View* renders the model into a form suitable for interaction, typically a user interface element. The *Controller* responds to events, typically user actions, and invokes changes on the model and perhaps the view. Views and controllers together comprise the user interface. Though MVC comes in different flavors, control flow generally works as follows:

1. The user interacts with the user interface;

2. The controller receives the raw input from the user interface and encapsulate the raw input as input events. The Command pattern (Gamma et al., 1995) is often used to encapsulate events. The controller sends the user input events to the model, and possibly to the associated view as well. In a non-distributed setting, sending an input event is done by calling the interfacing methods provided by the model. In a distributed setting, communication channels must be established for sending input events.

3. The model updates its internal state according to received input events.

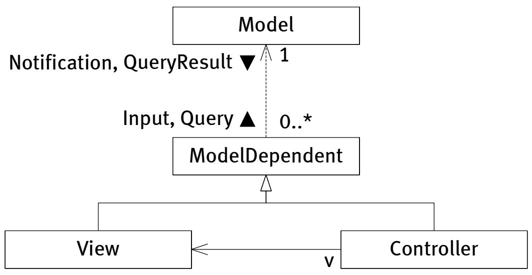
4. The model notifies the view and the controller about the internal state change. In a non-distributed setting, the model uses the Observer pattern to allow the controller and view components to subscribe their interests of the changes in the model and notify interested parties of a change to the observers. In a distributed setting, the Real-time Channel pattern can be used so that notifications can be sent through and filtered inside the connection channel according to the subscribed interests.

5. To render the interface, the view and the controller retrieve the data from the model. In a non-distributed configuration, this can be easily done with a straightforward function call. However in a distributed setting, this is often a two step process: the view or the controller sends the query request and the model returns the query result through a connection channel.

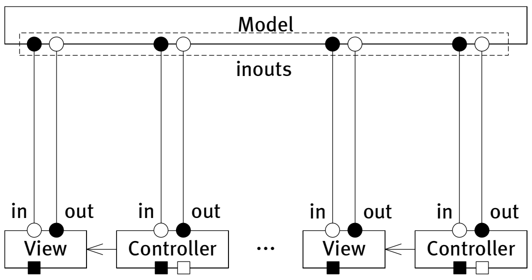
6. The user interface waits for further user interactions, which begins the cycle anew.

Whereas the view and controller components register themselves with the model and listen for changes, the model itself remains view and controller agnostic. The controller does not pass the model change to the view although it might issue a command telling the view to update itself.

Besides its wide use in non-distributed interactive architectures (JFC and MFC for example), MVC is also often seen in distributed applications, especially web applications. In a web application, the views are the HTML pages rendered by web



(a) MVC in UML

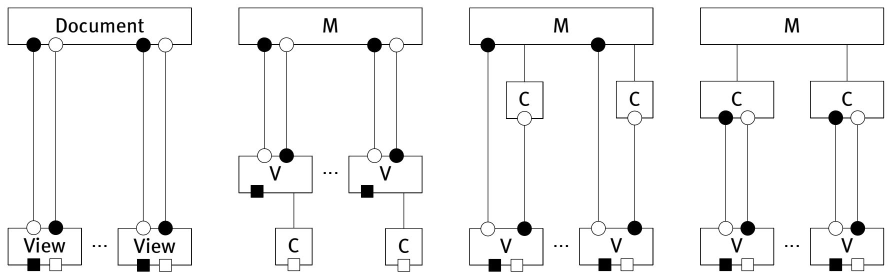


(b) MVC with channels

Figure 8.3: Distributed MVC

browsers and the code at the server side that generate the pages, and the controllers are the controlling components embedded in the HTML page, sending HTTP GET and POST requests to the model – the web service that implements the domain logic. While in these web applications, view and controller are often coupled locally at the client side together to comprise the interface, views are even split across the web client and the web server in web applications, which makes it unclear where the distributed boundary is between the view-controller coupled client and the model (web service). To simplify the specification, the splitting of the view components is ignored, and only the distribution of the model and its view-controller dependents are taken into account.

The overall static structure is shown in figure 8.3(a) in a standard UML diagram. *View* components and *Controller* components are paired locally and communicate with a remote *Model* component. *Views* and *Controllers* all *depend* on the *Model* component to update and render the interface, hence they have common attributes and behavior, which is modeled as the attributes and behavior of a *ModelDependent* component. Figure 8.3(a) shows the channel connections between the *Model* component and the *ModelDependent* components, in which not only are the symbols "●" and "○" used to indicate the data suppliers and consumers as introduced at the end of the previous chapter, but the symbol "■" is also used to indicate that the attaching component has a function of physically presenting data to the user, and the symbol "□" is used to indicate the function of capturing input from the user interface or the environment. See background material K for detailed Object-Z specifications.



(a) Document-View (b) M-VC (c) MC-V (d) MC-V

Figure 8.4: MVC variants

8.2.2 MVC variants

The MVC pattern has been just presented, with the configuration which would be applied in a distributed environment for interactive presentations. There are many variants, due to either different understanding of the roles of the controller component, or the difficulty of separating the controller from the associated view component. Three examples (figure 8.2.2) are given here.

Document-View The Document-View variant recognizes all three roles of Model-View-Controller but merges the controller into the view (figure 8.4(a)). The document corresponds to the model role in MVC. This variant is present in many existing GUI platforms. An good example of Document-View is MFC in the Microsoft Visual C++ environment(Kruglinski, 1995).

This variant is also used in distributed applications. For example, Okada and Tanaka (1997) used it for their distributed 3D graphics applications, with a Document-View structure named "Model-Displayobject". Interestingly, a previous attempt by the same authors for the same distributed applications used the conventional MVC structure (Okada and Tanaka, 1995). The reason of getting rid of the controller component in later systems was that "the role of the Controller could be included in the View part" and in practice "the View part directly handles user-operation events instead".

M-VC The M-VC variant is very similar to Document-View in the sense of isolating controller from the Model, but it does not combine the controller into the View (figure 8.4(b)). A controller in M-VC is only for taking user inputs and translate them into input events, then send to connected View. The controller does not have its interface of its own. Buttons and menu items can be controllers in MVC (figure 8.3 on the preceding page), but they have to be yet smaller M-VC agents in this variant. The granularity of this variant makes it more suitable for managing GUI components. Java Swing/glsjfc components are good examples of this variant. There JComponents are Views, and event handlers are controllers (Walrath et al., 2004).

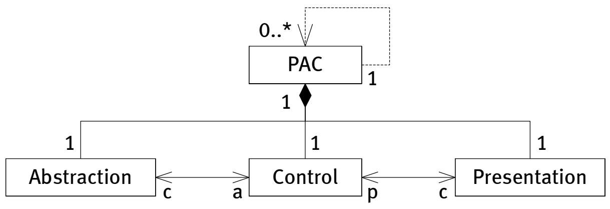


Figure 8.5: PAC in UML

MC-V Many distributed web applications advocate MVC as the architecture (Barrett and Delany, 2004; Bodhuin et al., 2002; GuangChun et al., 2003; Knight and Dai, 2002; Qiu, 2004; Wojciechowski et al., 2004), in which the controller component is actually a server side component that is tightly coupled with the model. Instead of being an interface component to get the input directly from the interface, the controller accepts the input events (HTTP GET and POST messages) from the web page (the view) rendered by a browser, and decides which functions or operations of the model should be invoked (figure 8.4(c) on the preceding page). In some implementations, the controller even coordinates the behavior of the view for the model (see figure 8.4(d) on the previous page). The controller falls into the Mediator pattern (Gamma et al., 1995) in this variant.

8.3 Distributed Presentation-Abstraction-Control

If the MVC architecture is applied at a system level, multiple View-Controller pairs need to communicate with the same *Model* component. When the system scales up, one model can hardly decompose the system complexity. More models are then needed. How these different models should be connected is not covered in the MVC architecture.

There are many attempts to improve the scalability of the MVC pattern, for example the Hierarchical-Model-View-Controller (HMVC) (Cai, Kapila, and Pal, 2000), in which the controller also takes the responsibility of communicating with other MVC components. However, when a controller does not handle the user input, but acts as a mediator between the model and the view, and even between the MVC agents, this kind of MVC becomes more PAC-alike than its name indicates.

Coutaz (1987) proposed a structure called Presentation-Abstraction-Control, which maps roughly to the notions of View-Controller pair, Model, and Mediator. It is referenced and organized in a pattern form by Buschmann et al. (1996): the PAC pattern “defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application’s functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction of the agent from its functional core and its communication with other agents.” Figure 8.5 shows the static structure of PAC in UML.

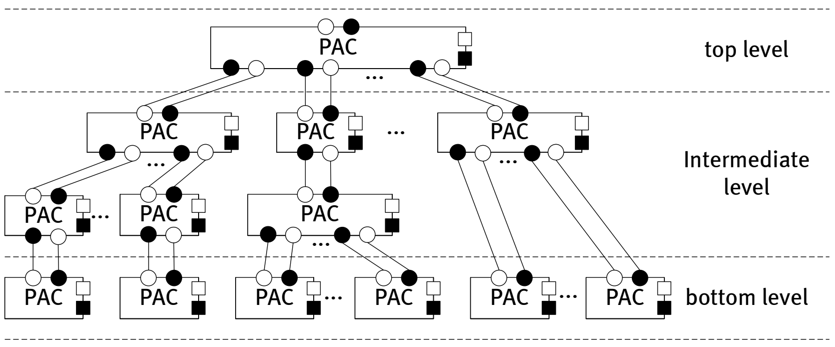


Figure 8.6: Distributed PAC in a hierarchy

8.3.1 Structure

This structure separates the user interface from the application logic and data with both top-down and bottom-up approaches (figure 8.6). The entire system is regarded as a *top-level* agent and it is first decomposed into three components: an *Abstraction* component that defines the system function core and maintains the system data repository, a *Presentation* component that presents the system level interface to the user and accepts the user input, and in between, a *Control* component that mediates the abstract component and the presentation component. All the communications among them have to be done through the control components.

At the *bottom-level* of a PAC architecture are the smallest self-contained units which the user can interact with and perform operations on. Such a unit maintains its local states with its own *Abstraction* component, and presents its own state and certain aspects of the system state with a *Presentation* component. The communication between the presentation and the abstraction components are again through a dedicated *Control* component.

Between the top-level and bottom level agents are *intermediate-level* agents. These agents combine or coordinate lower level agents, for example, arranging them into a certain layout, or synchronizing their presentations if they are about the same data. The intermediate-level may also have its interface *Presentation* to allow the user to operate the combination and coordination, and have an *Abstraction* component to maintain the state of these operations. Again, with the same structure, there is a control component in between to mediate the presentation and the abstraction.

The entire system is then built up as a PAC hierarchy: the higher-level agents coordinate the lower level agents through their *Control* components; the lower level agents provide input and get the state information and data from the higher level agents again through the *Control* components. This approach is believed more suitable for distributed applications and has better stability than MVC, and it has been used in many distributed systems and applications, such as Computer Supported Cooperative Work (CSCW) (Calvary, Coutaz, and Nigay, 1997), distributed real-time systems (Niemelä and Marjeta, 1998) , web-based applications (Illmann et al., 2000; Zhao and Kearney, 2003), mobile robotics (Khamis et al., 2003a,b), distributed co-design platforms (Fougeres, 2004) and wireless services (Niemela, Kalaoja, and Lago, 2005). However as far as known, there is no attempt so far to formally describe how the PAC works in a distributed setting.

To a large degree the PAC agents are self-contained. The user interface component (Presentation), the processing logic (Abstraction) and the component for communication and mediation (Control) are tightly coupled together, acting as one. Separations of these components are possible, but these distributed components would then be regarded as PAC agents completed with minimum implementation of the missing parts (examples will be given later). Thus the distribution boundaries remain only among the PAC agents instead of composing components. Based on this assumption, each component is formally described in background material L, modeling the communication among PAC agents with push style channels from the Channel pattern.

8.3.2 PAC Variants

There are also many PAC variants. Nigay and Coutaz (1991) propose a hybrid model called PAC-AMODEUS. In this model, the dialogue controller of the Arch model (Bass et al., 1991) is made of PAC agents. Other architectures have been proposed. For multiuser systems, PAC$*$ (Calvary et al., 1997; Khezami, Otmane, and Mallem, 2005; van Schooten, 2001) emphasizes the functional aspects of each component; for multimodal systems, AMF (Tarpin-Bernard and David, 1997; Tarpin-Bernard, David, and Primet, 1999) focuses on composition of the PAC agents; for distributed real-time systems, DPAC (Niemelä and Marjeta, 1998) integrates the runtime configuration support into a framework that supports distributed and asynchronous messaging; for consumer electronics and on-screen displays, Markopoulos et al. (1999) propose a scheme to distinguish "look" (*Presentation*) and "feel"(*Abstraction* + *Control*).

Despite the architectural amendments for different focuses, these variants are renditions of the basic structure that are formally specified in background material L: the *Control* acts as a mediator, separates the user interface *Presentation* from the function core *Abstraction*, and handles the communication among distributed PAC agents; a PAC agent comprises these three PAC components that locally form a hierarchical structure with other PAC agents through their *Control* components.

Lévy, Losavio, and Matteo (1998) and Lévy and Losavio (1999) propose the Broker pattern in instead of the Mediator pattern for communication. They argue that the Broker, in its indirect communication variant(Buschmann et al., 1996), is a special case of the Mediator, hence the usage of Broker instead of Mediator could take advantage form the distribution featured by Broker. The specification here agrees on the Mediator role of the *Control* component, but leaves the communication among distributed components to the connecting channels. The Channel pattern, however, uses producer and consumer proxies as output and input ports to connect the PAC agents, and these proxies actually fall into the Broker pattern. Note that these brokers are not PAC agent brokers as suggested by Lévy and Losavio (1999). Instead, they are producer and consumer brokers – the Broker pattern is applied at a more fine-grained level of each channel connection.

8.4 Comparing PAC to MVC

Based on the formal models of both MVC and PAC architectures in distributed settings, a concrete basis can be provided for language independent comparison. Both the MVC and PAC architectures are based on the concept of separating the functional core and data component of an interactive application from its interface components. The principle of separation allows these components to be developed separately and facilitates the reuse, but the way of separation differs in MVC and PAC. The primary differences between the PAC and MVC architectures are:

- the location of input and output;
- the communication among related agents and components;
- where the distribution boundaries are.

8.4.1 Coupling of input and output

When interacting with our physical surroundings, a "natural" interaction is expected: the output or the feedback of an interactive object should not be separated from the input or the action from the users. Wensveen (2005) proposes a tangibility approach in product design, trying to couple the action and the reaction as close as possible, in the aspects of time, location, direction, dynamics, modality and expression, to make the interaction with design objects more "natural".

In GUI design, direct manipulation (Shneiderman, 1997, 1981) is actually based on the same thinking. The input from the mouse, in the sense where the physical actions are taken, is (in location) not coupled with the reactions from the system (on-screen feedback). However when the user is getting used to the mouse to control the on-screen cursor, the mouse is a "ready-in-hand" (Dourish, 2001) tool that extends the physical capability of the user to the cursor. The "click" action is then felt to be happening at the place where the on-screen cursor is, not where the mouse is. Thus direct manipulation gives people a feeling of coupling thus a feeling of natural interaction. However direct manipulation of virtual objects using a mouse does not fall into the Tangible User Interface (TUI) framework of Ullmer and Ishii (2000).

TUIs give physical form to digital information, employing physical artifacts both as representations and controls for digital information, coupling the physical controls and physical representations with digital representations (Ullmer and Ishii, 2000). On the interface level, TUIs couple tightly the user control and the system feedback in physical means, hence bring more feeling of natural interaction to the users than direct manipulation on GUIs.

The basic concept of TUI fits very well with the vision of AmI, where the digital representations fade into the background and are embedded in our physical surroundings. The physical environment, and the physical objects which comprise this environment, become the interface to the digital data and functions. Interacting through tangible objects is one of the important ways to enhance the naturalness of interaction (Ducatel et al., 2001).

Ullmer and Ishii (2000) uses MVC to illustrate the separation of the GUI between the visual representation (or view) provided by the graphical display and the control capacity mediated by the mouse and keyboard of the GUI. For natural interactions through tangible objects, they present an alternate interaction model called MCRpd, for model-control-representation (physical and digital), where control-representation part is actually a coupled component to form the tangible interface. This view of coupled input and output is well reflected in PAC with the presentation component. An integrated component for both input and output would encourage the coupling of these two in the design of the virtual or physical objects.

Output behavior in the MVC architecture is dealt with by having the *Model* component handle the input events (*HandleInput*) from the *Controller* component and send appropriate notifications upon state change, and handle the data queries (*HandleQuery*) and send the query results to the *View* component. In PAC, input (*HandleInputData*) and output (*Present*) behaviors are coupled in the *Presentation* component. The MVC architecture models the output as to the depiction of abstract system states, whereas the PAC architecture models the output as encompassing both the creation and update of the interface the depiction of abstract system states. The *Presentation* component in PAC is essentially the entire user interface, composed of both the *View* and *Control* components.

8.4.2 Communication and control

PAC has a dedicated control component that mediates and manages the communication and that has no direct correspondent in the MVC architecture. In addition to its responsibility of the function and data services, the *Model* components in MVC must maintain the consistency between distributed MVC agents by passing messages through connecting channels, and the consistency between the distributed *View* and *Controller* pairs by monitoring the input events from all distant dependents and broadcasting change notifications through communication channels. However the MVC structure itself does not indicate how communication is handled between agents in an interactive system.

In the MVC specification, the *Model* component takes the control to decide what to do upon receipt of input events and the *Controller* component does not really control, but only act as an input component that takes the raw input from the user or the environment then transforms it into input events. There is no consensus in the design pattern society where the control really is and what is a *Controller* anyway [1] in MVC (see the variants in figure 8.2.2 on page 105). The *Controller* ends up being the most vague of all these components, and a satisfactory explanation can be hardly found – likely because MVC was designed for a very specific requirement in a specific environment (Smalltalk-80 and VisualWorks), and is difficult to reify it when it comes to describing distributed architectures such as web applications.

Comparing to MVC, the notion of control explicitly centralized in PAC. The PAC control hierarchy provides a useful mechanism for specifying relations between agents, because the hierarchy corresponds to the way in which the actors are distributed. Often however, the relationships are relatively simple and the approach

[1] A discussion is available at http://c2.com/cgi/wiki?WhatsaControllerAnyway.

adopted in the MVC architecture is sufficient. But for later distributed systems, the structure introduced by the PAC architecture provides easier maintenance than that provided by the MVC architecture.

8.4.3 Distribution

Although the *Presentation*, *Control* and *Abstraction* components of a PAC agent are locally coupled, the PAC agents are decoupled. In a PAC hierarchy, PAC agents can be loosely connected together through the communication channels using their *Control* components. Higher level agents can delegate their functions and interfaces to lower level agents. Consequently, making complicated, especially distributed, user interfaces based on simple components is more natural to PAC than it is to MVC because composition of the agents can be done without violating encapsulation. In this way, one can imagine that because of its flexibility of delegation and because of its decoupled nature, it is even possible to scale up PAC so that the various parts of PAC are completely separate processes. The properties are precisely what make PAC a good fit for distributed systems.

The formal specification of PAC looks more complicated than MVC, but PAC scales better. Adding a PAC agent, or a hierarchical collection of PAC agents, to a existing hierarchy is no more than connecting them to the communication channel that the parent PAC node is connected to. Removing is no more than disconnecting them. The specification of the PAC *Control* component is large, however it is because of the behavior of this component can be described at a fine-grained level. The specification of MVC is easier, but the communication between multiple MVC agents is simplified by sharing the same *Model* component, which would result in a very busy and centralized control. If multiple MVC agents are allowed to have their own *Model* components to build up a hierarchical structure as PAC has with the *Control* component, the specification of such a MVC structure would be more complicated, and it would also violate the principle of object oriented decomposition.

Moreover, the PAC pattern fits the problem of distributed user interfaces much better than MVC. First, it provides with a well-defined place to hook in the various infrastructure pieces that take care of the dimensions of the distribution and affect the user interface without exposing this functionality to the control logic of the application: the control component. This means that the control component can communicate with the location sensitivity system, the voice recognition engine, the speech synthesis engine and all the other sub-systems that are needed to control and produce the user interface without violating the separation of concerns among the abstraction, shielding the local logic and the presentation.

Both MVC and PAC styles are often used to structure interactive systems. Moreover, MVC is being employed as a reusable framework in commercial graphical toolkits. PAC instead, gradually being adopted by many research and industrial projects, has a major theoretical importance in research since it greatly favors the independence of communications among the agents, without loosing their control, and it seems very well adapted to be used to architecture distributed applications with multiple separated process. For this reason the design decision has been taken to use only the PAC style as the structure for the actors.

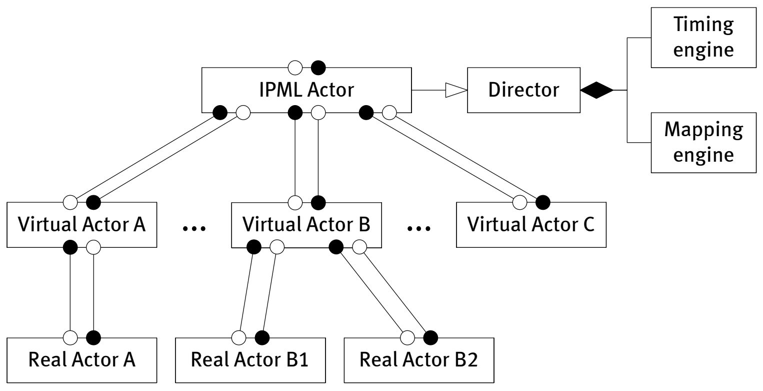


Figure 8.7: IPML system: an IPML actor

8.5 Actor: a PAC agent

An actor is basically a PAC agent, or to emphasize its interactive behavior, a PAC actor. It reacts on the user input events and scheduling commands, and takes actions to present media objects. Background material M describes an example implementation of an actor based on the PAC pattern, also using the patterns described in previous chapters.

8.6 IPML system: an IPML actor

The final IPML system is simply an IPML actor, or in other words, an *Actor* implementation that is capable of presenting the IPML scripts. IPML is a presentation description language that extends SMIL, describing the temporal and spatial relations among distributed actions on synchronizable content elements (more detail is available in the next chapter).

Note that IPML is an extension of SMIL, and a SMIL document is actually a composite content element by itself. Hence an IPML actor is first of all a SMIL player and it may present the contained content elements to its own *Presentation* component. What makes IPML superior to a SMIL player is the capability of distributed presentation, interaction and synchronization: It can delegate content presentations to other actors, synchronize the presentation actions of these actors, and propagate distributed user interaction events among these actors.

The overall structure of the IPML system (figure 8.7) is similar to the structure of the experimental StoryML system (see figure 3.9 on page 36), except that the scripting language is different and several architectural improvements have been made.

Comparing to the StoryML system structure, the final IPML design first of all abandons the notion of "physical actors", but uses "real actors" instead. The bottom level actors can be either physical devices or software agents. As long as it implements the interfaces and the functions of the *Actor*, it does not really matter at the system architecture level whether it is physically a real device, or virtually a software agent. The word "actor" is used not only implying the theater play metaphor, but also indicating these actors are implementations of the class *Actor* – PAC actors that perform to present content elements.

The second major difference is that in this architecture, only actors are distributed. Other functions, such as parsing an XML document, prefetching media objects and managing timelines are distributed with these actors, instead of realizing them as distributable agents. There are three reasons to make this decision:

1. These are mostly functions rather than objects. For example, prefetching has been designed as a function that should be implemented by every media object (see section F.12 on page 296), a centralized prefetcher is no longer necessary, but also violates the principle of encapsulation.

2. If the top level IPML system agent is implemented as an actor, objects such as the timeline controller and the XML parser are simply internal mechanisms of its *Abstraction* component. Whether to implement them as a separate PAC agent is not important, although doing so may enable visualization and interactive control of these mechanisms. Let's leave this as an option to implementation, rather than maintained at the system level as independent agents.

3. Encapsulating the scheduling functions internally, the IPML actor, is just yet another actor. This maximizes the extensibility and flexibility of the architecture: The entire system as an actor can join another IPML system in a distributed manner, without exposing the internal mechanisms to and interfering with each other.

In figure 8.7 on the facing page, the virtual actors are no longer managed by an "actor manager" as they are in the StoryML architecture, but rather, the IPML actor implements the role of a *Director*, which has a mapping engine, creates, manages and connects the virtual actors, and has a timing engine which schedules the timed actions for the delegating virtual actors. Depending on the physical configuration of the "theater" – the presenting environment, the mapping engine may also connect appropriate "real actors" to virtual actors, where the virtual actors keep the role of software drivers for the "real actors". It is then no longer necessary to have a distributed "actor manager" PAC agent between the IPML actor and the virtual actors – which may result in unnecessary network and management overload. The mapping engine may make use of distributed lookup and registration services such as Universal Plug and Play (UPnP) (Michael Jeronimo, 2003) and JINI (Edwards, 2000) to locate and maintain a list of "real actors", but this architecture leaves these possibilities open to the implementation of the mapping engine.

Looking back, in this part, an object-oriented formal approach is used to present the pattern-oriented architecture design of the IPML system, from action patterns that are used to present synchronizable content elopements, to channel patterns that are used to manage asynchronous communication among distributed objects, and finally in this chapter, to interface patterns for the actors. A high level view of the IPML system architecture is revealed: The IPML system is an IPML *Actor* that presents IPML scripts by delegating the interactive presentations to other virtual or real actors. An IPML actor also takes the role of a *Director*, reading and understanding the IPML scripts, finding appropriate actors for certain types of roles and actions, and scheduling action tasks for these actors. The next part focuses on the *Director*'s problems: the timing models and scheduling methods; and the mapping between the roles that are required in the script, and the actor cast that the *Director* has in a "theater".

Part III

Timing and Mapping

CHAPTER 9

Timing

An interactive play has been defined as a cooperative activity of multiple *actors* that take *actions* during certain periods of time to present content elements. An action, as the basic component of such an activity, has a time aspect per se. That is, a timing mechanism is needed to decide when the actor should commence the action, how long the action should take, and how the actions are related to each other in time.

The same phenomena exist in traditional multimedia systems. A multimedia system is characterized by integrated computer-controlled content generation, storage, communication, manipulation and presentation of independent time-dependent and time-independent *media objects*. Timing sits at the heart of multimedia systems, describing and managing the temporal relations between these media objects. Although the traditional media objects can not cover all the content elements that the actors need to take actions with, the timing issue is rather similar at least at conceptual level, if the difference of the managed objects is put aside. Therefore, the timing models from traditional multimedia systems can be useful for managing temporal relations of actions in IPML performances.

Many timing models and synchronization mechanisms have been developed for multimedia presentations during the last decades. The question is then which one fits better the needs of the theater play. The traditional treatments in multimedia systems rarely integrate the audience participation and interaction in the time model. For interactive theaters, it is necessary to model the audience participation and other events as a form of scheduling to provide a unified timing model for these commonly disparate features. Further, actors in the theater need to get their action commands and the related content elements in real time through different delivery channels. The timing model has to accommodate both reliable and unreliable delivery.

Although IPML has been presented as the scripting language, in which the SMIL timing model is used for describing the time relations between actions, the arguments behind the decision have not yet been covered. This chapter tries to cover the timing issues from the basis, by first giving an overview about the basic concepts and existing models, and analyzing their advantages and disadvantages. How the SMIL (Ayars et al., 2005) timing module covers general time relations will then be discussed. A run time synchronization engine is presented thereafter, addressing the above mentioned issues, providing a powerful, flexible and extensible framework for synchronizing the actions. This engine is used by the *director* to schedule the actions for the actors, no matter whether the actors are distributed over the network. However when the actors are distributed, time differences among their hosting devices and the network delay of scheduling commands can not be ignored. These distribution issues will be covered at the end of this chapter.

9.1 Background Concepts

9.1.1 Natural and synthetic temporal relations

Temporal relations between content elements can be divided into *natural* and *synthetic* relations (Little, 1995). In multimedia, natural temporal relations are implicit at the time of data capture, such as the interdependence between the audio and video tracks in the recording of a motion picture sequence. Synthetic relations are explicitly specified between independently captured or generated media objects. An example of this is the construction of a motion picture by compiling various sequences.

Similarly in interactive theaters, natural temporal relations may exist within the action, when the action is to perform a content element that is an autonomous behavior without external stimuli, for example, acting out the given lines. At the system level of action synchronization, the director assigns the actions to appropriate actors and schedules these actions in the play time according to the script; thus the temporal relations among actions are always synthetic.

9.1.2 Time-dependent and time-independent elements

A timing model defines how to coordinate and synchronize the actions over time. As said, an action is depicted by a content element. The term *content element* covers a broad range, including *time-independent* elements that have no intrinsic timing, and *time-dependent* types that are intrinsically time-based. The time dependency of the content elements influences significantly the action synchronization.

The content elements are often media elements. In multimedia, time-independent media elements are for example still images, HTML pages, vector graphics, whereas video, audio, and animations are time-dependent elements. In SMIL 2.0 , time-independent and time-dependent elements are respectively referred as *discrete* and *continuous* elements (Ayars et al., 2005).

Time-independent elements are all elementary synchronization objects because they are indecomposable for synchronization. After the time-independent elements have been started and before they are stopped, they will stay static and will not generate any time-based events. Furthermore, they can't start and stop by themselves since there are no internal time dependencies defined. The presentation of such an element has to be controlled by externally imposed time dependencies.

The major difference between time-dependent and time-independent elements is whether they have intrinsic timing. The same time dependency of the content element is imposed upon the corresponding action. A *time-independent action* only keeps the actor in a static situation, for example keeping a lamp on or off during the time. A *time-dependent action* often needs the actor to decompose it into more elementary actions and further schedule these actions in parallel or sequence by themselves. For example, flashing a light is decomposed to alternatively turning the light on and off.

9.1.3 Who has the time?

The timing aspect of a theater performance need to be supported by the authoring and performing systems, as well as the underlying timing models and engines. As Hardman et al. (1999) point out, there are different perspectives on the timing issue, from the authors of a play, the designers of the scripting model, the designers of the performing system (the theater), and of course, the audience of the performance.

From an author's perspective, timing aspects are important elements in the way the multimedia presentation is conceived and received and thus the author requires certain means of manipulating the timing within a performance. From a scripting model designer's perspective, temporal aspects of a performance are about the composition of the actions with some structures that are not only easy for the authors to express timing relations, but also easy for the performing system to understand. From a performing system designer's perspective, different synchronization strategies and techniques are needed for different types of actions, for example, those with or without audience interaction, and those with linear or non-linear navigation. The system designer also needs to decide how to deal with situations such as network delay and device variation. The content is finally presented to the audience, who should receive and perceive the performance in a simple timeline without knowledge of the underlying temporal constructions.

Hardman et al. (1999) present a useful taxonomy for these time concepts. There are four types of time when constructing and conducting a performance[1]:

Content element time: is the duration of (part of) the content element included in a play. If there are no further temporal transformations, the media element time is the length of time it would take to be performed on an ideal system. The content element time can only be manipulated in the script, unless the author edits the element itself. Some content elements are distributed over and streamed from a network, and the durations of these elements can not be determined at the authoring stage until they have been presented at the client side and explicitly ended from the server side. A good example of such elements is a live audio or video program. Some other content elements have no intrinsic timing, such as images and text, the durations of which depend on how it is performed or presented. For example if the text is displayed on a screen, the duration is often treated as infinite. But if the text is going to be presented through a TTS engine, the duration depends on the synthesizing mechanism. The designers of the timing model and the presentation system have to take all these different elements into account.

[1]Hardman et al.'s taxonomy is adapted for the play metaphor. In the original taxonomy for the time concepts in multimedia systems, the adapted time and action time were called *rendered time* and *runtime* respectively.

Script time: is the relative time stored in the script describing when the content elements should start and stop. The author can assign and manipulate the script time, in terms of a timing model, or indirectly using an authoring tool and then converting to the timing model. The presentation system should retrieve the content elements included in the document in due time and comply with the temporal relations specified by the authors to construct the presentation.

Adapted time: is the result of adapting the script time to the performing system. The script may provide alternative content elements for different client system configurations. After the authoring stage and before the final performance, the performing system has to decide which content element fits best to the client environment, which will result in different timing schemes on different systems (again the example of the text being rendered by different means can be used). If no choices are provided in the script, the adapted time is the same as the script time.

Action time: is the time when the performance takes place in real time, during which the audience can interact with the performance and, possibly, influence the actual timeline of the performance. The presentation time can also be affected by network delay when the media elements are distributed over the network.

Each time type has its own important role in the synchronization. For example, in a script, it could be specified that an audio element should start right after a video element has started. If the action time of the video element is delayed, so is the audio. But if both the audio and the video are specified to be started 5 seconds after the beginning of the entire play, the delayed video should not prevent the audio from starting immediately at its due time, and the video must catch up with the audio by for example dropping a certain number of frames.

9.1.4 Synchronization reference model

A reference model is needed to understand the various requirements for multimedia synchronization, identify problems and compare existing solutions for multimedia systems, and structure the runtime strategies that support the execution of the synchronization. A layered reference model (figure 9.1(a) on the next page for multimedia synchronization is introduced by Meyer, Effelsberg, and Steinmetz (1993) and further developed by Blakowski and Steinmetz (1996).

This reference model provides four layers of abstraction through which a multimedia application can access synchronization service. Each layer implements synchronization mechanisms which are provided by an appropriate interface. These interfaces can be used to specify or enforce the temporal relationships. Each defined interface services can be used by the next higher layer to implement the higher level interface. At the media layer, a presentation operates on a single continuous media stream, which is treated a sequence of Logical Data Units (LDUs), such as video frames and audio samples. If this layer is used by the presentation, the presentation itself is responsible for the data units to be played back at a correct interval, enforcing the intra-media synchronization.

The stream layer operates on continuous media streams, as well as on groups of media streams. Typical operations invoked by an presentation are for example *start*

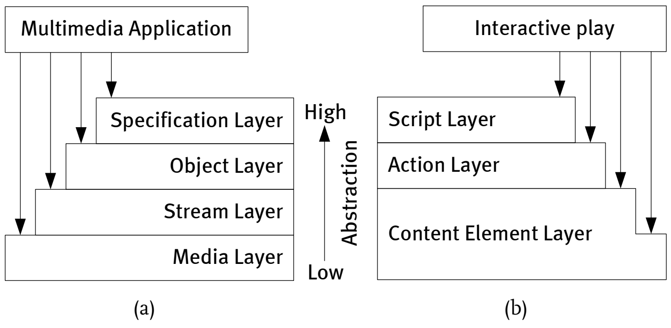


Figure 9.1: Four-layer reference model

and *stop* a stream or a group of stream. The object layer operates on all types of media and hides the difference between the time-independent and time-dependent media. At the specification layer, the spacial and timing relations between the objects provided by the object layer are defined, such as a video object to be presented in parallel with an audio object, and with a sequence of subtitle text objects. Spatial relations such as a “picture in picture” presentation are also defined in this layer, but the spatial relations are not going to be covered here.

Two types of temporal media synchronization are distinguished in multimedia applications: inter-media and intra-media synchronization. The first refers to synchronization among media streams (e.g. video and audio in lip-synchronization), while the second refers to synchronization within a single stream (e.g. video frames to be presented in synchronization with time intervals for continuous playing back at specified frame rate).

For interactive plays, this reference model is adapted to three layers of content element, action and script (figure 9.1(b)), in which the content element layer contains both the stream layer and the media layer in figure 9.1(a).

In the content element layer, a content element encapsulates its internal synchronization mechanisms to provide proper synchronization behavior in content element time for the action layer. The concept of content element is more general than the concept of media objects. A content element can be a composite media object, for example a movie clip with internally synchronized video and audio streams, hence it may cover both the media layer and the stream layer in the reference model. Let’s leave the internal synchronization issues of a content element to the rendering platform and focus on the inter-media or inter-action synchronization.

The action layer wraps the content element up and hides the differences between time-dependent and time-independent elements, providing the script layer with unified action services, for example, preparing, starting and stopping the content element at a given time. The time can be specified in the script in a relative manner in script time, and during the performance, the time should be first mapped to the adapted time according to the configurations of the system, and then mapped to the absolute action time for scheduling the action.

The script layer specifies the timing relations among the actions using timing models. These models use relative time instants, or interval based relations. Some may use both. During the performance, no matter whether the timing model is instant or interval based, the actual action time of these actions must comply with the relations defined in the script.

9.2 Timing models

In multimedia systems, two representations of temporal relations can be indicated. These are based on *instants* and *intervals*. A time instant is a zero-length moment in time, such as "9:30am". By contrast, a time interval is defined by two time instants and therefore, their duration (e.g., "10s" or "9:30am to 9:40am").

9.2.1 Instant based models

Let's start from the concept of time instants as an axiom for further discussions: an *instant* is a durationless moment in time, or more precisely, an instant is a piece of time whose begin and end are not distinct in the context of the consideration.

The idea that time instants should be the primitives in a temporal relations has been influenced by *Physics*. In *Physics* it is a common practice to model time as an unbounded, ordered continuum of instants that is structured as the set of the real numbers (Whitrow, 1980), and hence has an unbounded dense linear relation (i.e., $<$). Not only researchers from the Artificial Intelligence (AI) community are in this tradition (Galton, 1990; McDermott, 1982; Shoham, 1987), but also researchers in temporal databases use this assumption, as it is, for example, described by McKenzie and Snodgrass (1991). There is a powerful argument for this view, that is, instants are important for modeling continuous change (Galton, 1990; McDermott, 1982). For example, if one throws a ball into the air, then a point of its trajectory depends on an instant. In order to describe the dynamics of classical mechanics, one needs concepts such as derivatives, and hence the theory of real numbers (Newton's $F = m\frac{d^2x}{dt^2}$). In addition, if intervals become necessary, they can be introduced as ordered pairs of instants, as it is done, for example, in (Ladkin, 1987; McDermott, 1982). One may refer to (Broxvall and Jonsson, 1999) for further considerations of the instant based models over partially ordered (e.g., branching) time.

Let's consider the basics of instant based models. Formally, an instant based model is defined over a structure $\langle \mathbb{T}, \prec \rangle$, where $\mathbb{T}$ denotes a non-empty set of instants, and $\prec$ defines the ordering relation between two instances.

A temporal structure may or may not have on or more of the following properties:

Discreteness: An instant is discrete in a temporal structure, if, along any path in the structure which includes that instant, the instant has a "closest" instant on each side, unless it has no instants on that side:

$$\forall t, t_1 : \mathbb{T} \bullet t \prec t_1 \Rightarrow (\exists t_2 : \mathbb{T} \bullet t \prec t_2 \wedge (\forall t_3 : \mathbb{T} \bullet \neg (t \prec t_3 \prec t_2)))$$
$$\forall t, t_1 : \mathbb{T} \bullet t_1 \prec t \Rightarrow (\exists t_2 : \mathbb{T} \bullet t_2 \prec t \wedge (\forall t_3 : \mathbb{T} \bullet \neg (t_2 \prec t_3 \prec t)))$$

A temporal structure is discrete if all instants in it are discrete.

Density: A temporal structure is dense if between any two comparable instants there is a third instant:

$$\forall t, t_1 : \mathbb{T} \bullet t \prec t_1 \Rightarrow \exists t_2 : \mathbb{T} \bullet t \prec t_2 \prec t_1.$$

In particular, density is valid if instants are modeled as real numbers ($\mathbb{R}$) or rational numbers ($\mathbb{Q}$), but not if as integers ($\mathbb{Z}$) or natural numbers ($\mathbb{N}$).

Ordering: The relation $\prec$ constrains the set $\mathbb{T}$ with a strict partial order if it satisfies the properties of

$$\begin{array}{lr} \forall t : \mathbb{T} \bullet \neg (t \prec t) & [\,\mathsf{Irreflective}] \\ \forall t, t_1, t_2 : \mathbb{T} \bullet t \prec t_1 \wedge t_1 \prec t_2 \Rightarrow t \prec t_2 & [\,\mathsf{Transitive}] \\ \forall t, t_1 : \mathbb{T} \bullet t \prec t_1 \Rightarrow \neg (t_1 \prec t). & [\,\mathsf{Antisymmetric}] \end{array}$$

Linearity: may be imposed to time to force instants to be arranged in a single line:

$$\forall t, t_1 : \mathbb{T} \bullet t \prec t_1 \vee t = t_1 \vee t_1 \prec t$$

that is, in a linear temporal structure, two instants must be either before or after each other, or they are simultaneous.

Unboundedness: A temporal structure is *unbounded* if

$$\nexists t_1, t_2 : \mathbb{T} \bullet \forall t : \mathbb{T} \bullet t_1 \preceq t \preceq t_2.$$

Otherwise the structure is either *left-bounded* such that nothing has happened before a starting point:

$$\exists\, begin : \mathbb{T} \bullet \forall t : \mathbb{T} \bullet begin \preceq t,$$

or *right-bounded* such that there is an end towards future:

$$\exists\, end : \mathbb{T} \bullet \forall t : \mathbb{T} \bullet t \preceq end.$$

Different time structures have been used. In chapter 4, time is modeled as rational numbers[2] ($\mathbb{Q}$) with which the authors of the script may specify the begin and end time and the duration of an action in various date and time formats (e.g. $1000.0000002ms$). The begin time can be relatively negative, so that the action will start from a point in the middle of its content element time (e.g. begin = "$a.click - 1000.0000002ms$"). The time there is dense, linear and unbounded, which conveniently reflects our normal perception of time. However in chapter 6, $\mathbb{T}$ is modeled as $\mathbb{N}$ that is discrete, linear and left-bounded. The value of $\mathbb{T}$ there starts from a fixed point in time, progressively advancing in discrete steps, which fits better the internal representation of the clock time in the processing unit. Although time structures in the specification script and the runtime scheduling engine are different, the mapping between these two structures is straightforward and most of the platforms provide a convenient mapping Application Programming Interface (API).

Next two instant based timing models, timeline and temporal point net, are presented as examples. They are about ways of specifying temporal relations between content element time instants and script time instants[3].

[2] Par abus de language these numbers are called “real” numbers as well.
[3] Action time (runtime) instants are linearly ordered anyhow, of course.

Timeline

A common model employing instant-based temporal intervals is the timeline, in which instants, representing the temporal boundaries of media objects, are ordered along a time axis.

Let $\mathbb{T}$ be a set of instants, and $<$ a strict total order on the set $\mathbb{T}$, then for any two temporal instants, represented by t_1 and t_2, there are three possible temporal relations in a linear structure:

- $t_1 < t_2$ (t_1 before t_2)
- $t_1 = t_2$ (t_1 simultaneous to t_2)
- $t_1 > t_2$ (t_1 after t_2)

These are the three basic relations between time instants which allow for expressing eight disjunctive relations between time instants; for example, the disjunction of $<$ and $=$, denoted by $\leqslant$, or the disjunction of $<$ and $>$, denoted by $\neq$, are such disjunctive relations. These relations correspond to different degrees of knowledge on the involved pairs of instants.

Many projects and products use timeline based time models. In the early stage of this project, the StoryML conceptual model (see chapter 3) uses an implicit timeline to specify when storylines and user interactions should start and stop. In the Athena Muse project (Hodges, Sasnett, and Ackerman, 1989), a global timeline is used to attach all media objects to a time axis that represents an abstraction of real-time. With some modifications, this kind of strategy is also used in the model of active media (Tsichritzis, Gibbs, and Dami, 1991), Apple QuickTime (Towner, 1999) and Macromedia Director (Persidsky and Schaeffer, 2003). A world time is maintained, which is accessible to all objects, each object can map this world time to its local time and moves along its local time axis. In presentation time, re-synchronization with the world time is required when the difference between world time and local time exceeds a given limit.

The project Athena (Hodges et al., 1989) and the HyTime standard (Erfle, 1994; Goldfarb, 1991) use multiple virtual time axes, a generalized version of the timeline approach. With this approach, it is possible to use several virtual axes to create a virtual coordinate space and specify the temporal relations with user-defined measurement units.

Timeline based models relate any two instants by one of the instant relations $<, =, >$. Owing to the fact that all instants are totally ordered along the time axis, it is impossible not to define a relation between any two events. Since the relations can only be defined based on fixed instants of time, problems arise if the objects have unpredictable durations. This restriction makes the model somewhat inflexible (which is precisely what was found out in the evaluation of StoryML). In other words: the timeline model forces the author to commit the sin of over-specification.

On the other hand, the timeline approach allows a very good abstraction from the internal structure of the objects and the nested structures. It is also very intuitive and easy to use in authoring situations.

Temporal point net

Temporal point nets used in the Firefly multimedia document system (Buchanan and Zellweger, 1993, 2005, 1992) are an instant-based approach that defines relations between interval end points for temporal composition (figure 9.2). Temporal constraints are ordering relations that can be placed between pairs of events in one or more media items. The relations are based on the events which establish temporal equalities ($=$) and inequalities ($<, >$) between instants (interval end points) in a presentation. For example in figure 9.2, the object *Audio A* must start simultaneously when the *Image* object ends, and when *Audio A* ends, the object *Audio B* must start at the same time. *Audio B* should start earlier than the *Video* object.

In contrast to the timeline approach it is no problem to let the temporal relation between two instants be unspecified. This makes the temporal net more flexible than the timeline (Buchanan and Zellweger, 1993, 2005; Wahl and Rothermel, 1994).

However, the use of temporal point nets is less intuitive than the timeline approach and it may result in complicated, unstructured graphs. In addition to that, their use may lead to an inconsistent specification in which contradictory conditions are specified for intervals. In this case, a verification algorithm (called sometimes a temporal formatter) is needed to check the consistency.

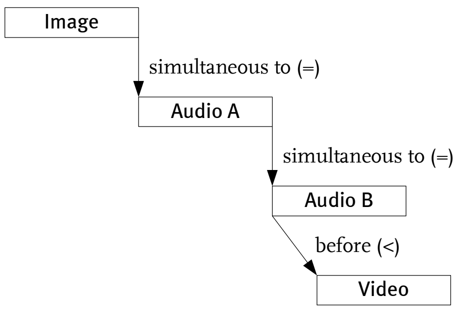


Figure 9.2: Temporal point net

9.2.2 Interval based models

Before discussing the interval based time models, let's first introduce a definition of *time interval*:

Let $\mathbb{T}$ be a partially ordered set, $\leqslant$ be the partial order on the set $\mathbb{T}$, and t_1, t_2 be any two elements (time instants) of $\mathbb{T}$ such that $t_1 \leqslant t_2$. The set $\{x \mid t_1 \leqslant x \leqslant t_2\}$ is called an *interval* of $\mathbb{T}$ denoted by $[t_1, t_2]$.

Any interval $[a, b]$ has the following properties:

- $[t_1, t_2] = [t_3, t_4] \Rightarrow t_1 = t_3 \wedge t_2 = t_4$
- $t \in [t_1, t_2] \wedge [t_1, t_2] \subset [t_3, t_4] \Rightarrow t \in [t_3, t_4]$
- $\#([t_1, t_2]) \geqslant 1$

Temporal intervals are defined by their end points (here represented by t_1 and t_2). Whenever $\mathbb{T}$ is embedded in $\mathbb{N}$, $\mathbb{Q}$, or $\mathbb{R}$, the length of such a temporal interval is $t_2 - t_1$. Given an interval a, let's write δ_a to denote its length.

Allen (1983) introduces the thirteen basic relations that can exist between temporal intervals and these relations are further discussed by Allen and Hayes (1990), and Allen and Ferguson (1994). Figure 9.3 depicts seven of these relations based on a timeline. The remaining six can be constructed by taking the inverse of each relation except "equals" (Little, 1995; Little and Ghafoor, 1993). For example, the inverse relation of "before" is "after (or "before^{-1}"), where "a before b" is the same as "b after a". Temporal intervals can be used to model multimedia presentations by letting each interval represent the presentation time of some multimedia content element, such as a still image or an audio segment.

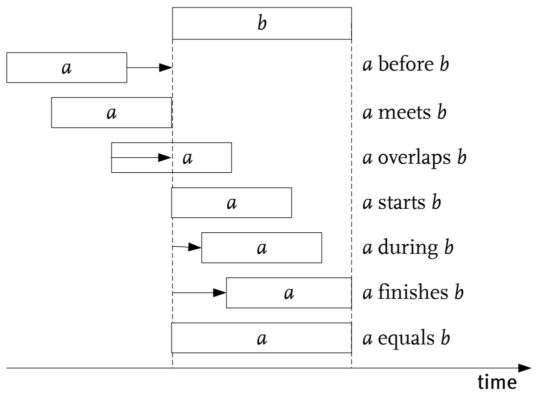


Figure 9.3: Temporal Relations

Both language-based (including scripting) and flow graph-based approaches have been proposed for abstracting the interval relations via a representational scheme. Among others, Object Composition Petri Net (OCPN) is an example of the graph-based approach, while SMIL is an example of the scripting approach.

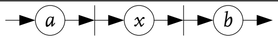
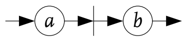
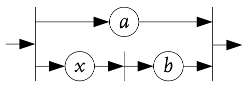
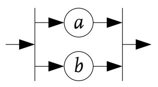
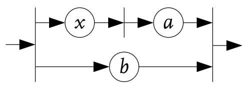
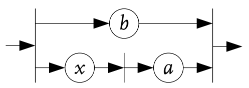
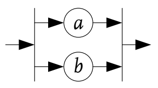
Object Composition Petri Net

Petri nets are commonly used to model concurrent systems. Extensions have been made to Petri nets to improve their functionality in representing multimedia synchronization. Little and Ghafoor (1990) propose Object Composition Petri Net (OCPN), a formal specification and modeling technique, for multimedia composition with respect to inter-media timing, based on the logic of temporal intervals and Timed Petri Nets. It allows the specification of synchronization requirements of complex structures of temporally related objects in a database, and the retrieval of media elements from the constructed database in a manner which preserves the temporal requirements of the initial specification.

A Petri net is a network of a set of *transitions* (bars), a set of *places* (circles), and a set of directed arcs. OCPN augments the conventional Petri net model with values of time, as durations, and resource utilization on the *places* in the net. In OCPN, each *place* (circle) in the net has a duration and represents the playback process of a media item or a delay. *Transitions* (bars) represent synchronization, as usual.

Equipped with interval constraints, the OCPN model has been proved capable of specifying any arbitrary temporal relationship and it may be applied to both stored-media applications and live media applications (Little and Ghafoor, 1991, 1990). It can easily cover the 13 temporal relations – given any two atomic temporal intervals, there exist an OCPN representation for their relationship in time (table 9.1).

Table 9.1: Temporal relations in OCPN

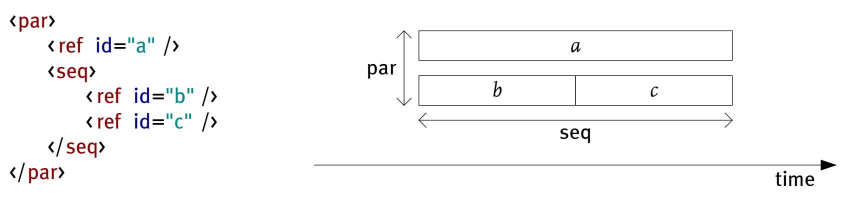
Temporal relation	OCPN representation	Interval constraints
a before *b*		$\delta_x \neq 0$
a meets *b*		
a overlaps *b*		$\delta_x \neq 0 \wedge \delta_a < \delta_x + \delta_b$
a starts *b*		$\delta_a < \delta_b$
a during *b*		$\delta_x \neq 0 \wedge \delta_x + \delta_a < \delta_b$
a finishes *b*		$\delta_x \neq 0 \wedge \delta_b = \delta_x + \delta_a$
a equals *b*		$\delta_a = \delta_b$

The eXtended OCPN (XOCPN) (Woo, Qazi, and Ghafoor, 1994) can additionally specify the resource management and communication functions in control places. Prabhakaran and Raghavan (1993) suggest a Dynamic Timed Petri Nets model which can be adopted by OCPN to facilitate modeling of multimedia synchronization characteristics with dynamic user participation. Guan, Yu, and Yang (1998) propose Prioritized Petri Net (P-net) and an enhanced version (EP-net) (Guan and Lim, 2002), and apply them to distributed multimedia synchronization, using their version of Distributed OCPN (DOCPN) model. Using the DOCPN model, operations among distributed computer sites can be coordinated (Guan and Lim, 2002; Guan et al., 1998; Shih, Keh, Deng, Yeh, and Huang, 2001), which is very interesting for the purpose of synchronizing distributed actions.

SMIL Timing Module

Often an interval based modeling uses only the parallel (*equals*) and sequential (*meets*) relations out of the thirteen temporal relations. By restricting temporal composition operations to these relations, most temporal interactions can be specified. This approach has been used by Poggio et al. (1985) in the development of the Command and Control Workstation Project (CCWS), by Postel et al. (1988) and Reynolds et al. (1985) for grouping presentation elements under *sequential* and *simultaneous* property, and by Ravindran and Steinmetz (1996) using AND-OR graphs and an occurs-after relation to specify timing precedence. To completely represent the 13 temporal relations, often a time object is introduced in these models to be inserted into the vacant time interval between any disconnected objects. By introducing such a time object, Fung and Pong (1994) prove that these 13 relations can be refined as two parallel primitives *starts_with* and *end_with*, and a sequential primitive *follow_by*.

SMIL is another example. The temporal relations in a SMIL document can be defined using the SMIL timing and synchronization module (Ayars et al., 2005; Rutledge, 2001), which in the following is described on a general level. The model is based on the “meets” and “equals” relations. These relations represent sequential and parallel playback of media items, respectively, and are represented by the SMIL synchronization containers seq and par. Figure 9.4 shows a SMIL fragment and its timeline interpretation (note that ref is a SMIL generic media reference).



```
<par>
    <ref id="a" />
    <seq>
        <ref id="b" />
        <ref id="c" />
    </seq>
</par>
```

Figure 9.4: Timeline interpretation of a SMIL fragment

This type of temporal specification approach is also known as a hierarchical control flow-based specification (Bulterman and Hardman, 2005), as the synchronization behavior can be seen as a flow of control through a hierarchical tree structure.

The SMIL timing and synchronization module is not limited to a basic hierarchical specification, however. The model is made more expressive by the synchronization attributes such as begin, dur and end which can be specified for each synchronization element. The dur attribute specifies an explicit duration for a synchronization element, begin and end are used to specify synchronization behavior for the end points of an element. End point synchronization can be used, for example, to synchronize the begin or end time of an element to some events in another element and to user interaction events. Examples of such events are an media element e being started for 5 seconds (“$e.beginEvent + 5s$”) and a button btn on the screen being pressed (“$btn.activateEvent$”). The addition of end point based synchronization attributes makes the SMIL timing and synchronization module a hybrid between an interval-based and an instant-based model.

Other additional attributes such as endsync, repeatdur and repeatcount can be added to the synchronization elements. For example the attribute endsync that can be added to a par container to control the implicit duration, as a function of the children. The endsync attribute is particularly useful with children that have "unknown" duration.

It is not practical to list all SMIL timing and synchronization elements and attributes here. But with the par and seq elements and the attribute begin, the timing SMIL is already powerful enough to cover all thirteen temporal relations (table 9.2).

Table 9.2: Temporal relations in SMIL

Temporal relation	SMIL specification	Constraints
a before *b*	`<seq>` `<ref id="a" />` `<ref id="b" begin="t" />` `</seq>`	$t > 0$
a meets *b*	`<seq>` `<ref id="a" />` `<ref id="b" />` `</seq>`	
a overlaps *b*	`<par>` `<ref id="a" />` `<ref id="b" begin="t"/>` `</par>`	$t > 0 \wedge a.dur < b.dur + t$
a starts *b*	`<par>` `<ref id="a" />` `<ref id="b" />` `</par>`	$a.dur < b.dur$
a during *b*	`<par>` `<ref id="a" begin="t"/>` `<ref id="b" />` `</par>`	$t > 0 \wedge t + a.dur < b.dur$
a finishes *b*	`<par>` `<ref id="a" begin="t"/>` `<ref id="b" />` `</par>`	$t > 0 \wedge t + a.dur = b.dur$
a equals *b*	`<par>` `<ref id="a" />` `<ref id="b" />` `</par>`	$a.dur = b.dur$

Note that in table 9.2, the constraints can be enforced either implicitly by the involved ref elements having the required duration, or explicitly by adding a dur attribute to these elements. For example, "*a* starts *b*" can be specified explicitly in SMIL as

```
<par>
    <ref id="a" dur="δa" />
    <ref id="b" dur="δb" />
</par>
```

where $\delta_a < \delta_b$.

9.2.3 Timing models for authoring and directing plays

Given that many timing models are available, it is necessary to decide which timing model is to be used for authoring the plays and directing the performances.

Timeline based instant models use a time axis (or multiple time axes or time branching) as the main method of structuring the temporal positioning of content elements. This approach is often used in the editors for time-dependent and composite content elements (for example MPEG-2 movies) where the start times and the durations of the included items are explicitly available, and in principle can also be manipulated directly.

However when the temporal characteristics of the time-dependency are not completely known or can not be explicitly specified, timeline based models can hardly handle conditional activations and asynchronous behavior, nor can they deal with adaptive inter-media or inter-action constructions. For interactive and adaptive plays, the timeline metaphor is not powerful enough for either authoring or directing.

Graph based models, including both instant and interval based graph models, provide abstractions that incorporate powerful temporal compositions and allow grouping and nesting these compositions to reflect complex narratives. Controlling nodes and arcs can be added to the graphs such that the models can be flexible enough to support arbitrary temporal structures. Graph based models are often based on a formal mathematical model that can be easily implemented as internal representations for scheduling the presentations, and in our case, directing the performance.

Nevertheless, when graph based models are used for authoring, there is considerable overhead in specifying all of the constraints associated with the presentation states of the content elements, and the flexibility often results in a loss of focus for the authors who are not good at, or less interested in formal specification and formal analysis other than simply specifying the narrative paths.

Comparing the script based models to graph based models, for example the SMIL timing and synchronization module in table 9.2 on the preceding page to the OCPN model in table 9.1 on page 127, one may find that the script models are essentially similar to the graph models in terms of their flexibility and their expressiveness. The script models are likely to be more flexible due to the direct use of a scripting language, so that the authors are not restricted to the actions supplied by an authoring system for manipulating the timing model. However when more flexibility is handed over to the authors, more structural inconsistency and more errors may occur. Without further supporting tools, scripting is often a tedious and low-level method for specifying a multimedia presentation.

In spite of the disadvantages of the script models, the most flexibility from these models is needed for specifying complex narrative structures and detailed user interactions in interactive plays. Let's leave the problems of the tedious, error prone scripting process to dedicated authoring tools, and concentrate only on interpreting such a script model to an internal representation for scheduling.

Many scripting languages are available for specifying multimedia presentations. For example, Videobook timed scripts (Ogawa, Harada, and Kaneko, 1992) provide hypertext-based nodes to present text, graphics and audio visual data synchronously and control the data sequence along a timeline; Harmony hypertext scripts (Shimojo et al., 1991) uses links to specify the timing relations between media nodes; Nsync (Bailey et al., 1998) defines a declarative synchronization language that supports the specification of synchronous and asynchronous timing relationships using conjunctive and disjunctive operators based on branching timelines. Some scripting languages are powerful and flexible enough for our purpose, however, it is preferred to use a script language that is more open and extensible, and possibly supported by open standards and tools. None of the aforementioned languages falls into this category.

XML based languages fit this aim very well, and during the early phase of this project, XML-based StoryML was designed, based on a timeline model to describe timing relations. However as already found out, StoryML was only suitable for linear or, at best, branching structures. The timeline based model was not powerful enough for complex narratives. During the course of this project, SMIL as a W3C recommendation for synchronized multimedia, especially for distributed multimedia over the web, was gradually getting more acceptance and support from the industry. The SMIL timing and synchronization module, as briefly depicted in this section, can easily cover all the interval-based time relations, furthermore, its event and link model supports user interactions and complex narrative structures, and works seamlessly together with the timing model. Hence it was decided to extend SMIL for interactive plays, including its timing and synchronization module as the basis for specifying timing relations. This timing model was referred to as IPML timing model hereafter.

A timing engine is then needed for the director that takes the IPML timing specification as input, schedules the specified actions and issues the scheduling commands to the actors. W3C has a recommendation of complete Java bindings for the SMIL 1.0 DOM (Le Hégaret and Schmitz, 2000). Its object-oriented design can be used as the basis for interpreting a IPML script into an object-oriented representation. Nevertheless, it is just a static representation of the hierarchical document structure and is far from a timing engine. Since the IPML timing model is basically interval based, and the graph based formal models can be better representations for scheduling the presentations, let's develop a timing engine based on a graph based interval model. The general notion of timed Peri nets, especially the OCPN model, have been explored extensively in the research literature, which provides a solid basis, and in practice, many examples to start from. However, the OCPN model also has limitations in dealing with interaction events and network delays.

The next section starts from the timed Petri nets and the OCPN model, investigates their limitations, and proposes new model for the IPML director.

9.3 Action Synchronization Engine

Before the synchronization engine is proposed, let's first have a look at the basic concepts of Petri nets and OCPN.

9.3.1 Petri nets

Petri nets were developed originally by Carl Adam Petri (Petri, 1962, 1966), and were the subject of his dissertation *Kommnunikation mit Automaten*. Since then there has been a steadily increasing interests in Petri nets because of their capability of representing both concurrency and nondeterminacy (Best and Devillers, 1987). Thirty years of theoretical and practical work and several thousand research papers prove that Petri nets are one of the most useful means for modeling concurrent processes.

The Petri net is defined as a bipartite, directed graph $G = (PL, TR, AR)$ where

$$\begin{aligned}
&PL = \{pl_1, pl_2, \ldots, pl_n\}, \text{ where } n \geqslant 0 && [\textit{places}]\\
&TR = \{tr_1, tr_2, \ldots, tr_m\}, \text{ where } m \geqslant 0 \wedge PL \cap TR = \varnothing && [\textit{transitions}]\\
&AR : I \cup O, \text{ where} && [\text{directed arcs}]\\
&\qquad I = PL \leftrightarrow TR && [\textit{input}\text{ arcs}]\\
&\qquad O = TR \leftrightarrow PL. && [\textit{output}\text{ arcs}]
\end{aligned}$$

A marked Petri net $G_M = (PL, TR, AR, MA)$ includes a marking MA which assigns tokens to each place in the net: $MA : PL \rightarrow \mathbb{N}$. MA is a total function that maps places to natural numbers ($\{0, 1, 2, \ldots\}$), which means a place can be marked with one or more tokens, or no token at all. The behavior of the Petri net is governed by a set of *firing rules* that allows the tokens to move from one place to another:

1. A transition is enabled when all the input places that are connected to it via an input arc have at least one token;

2. Firing of a transition removes a token from each input place and puts a token in all of its output places.

9.3.2 OCPN

For the simple Petri nets, there is no notion of processing time in either places or transitions. The time from enabling a transition to firing is unspecified and indeterminate. For a representation of the synchronization of multimedia entities, Petri nets must be extended.

The OCPN model augments the conventional Petri net model with values of time, as durations, and resource utilization on the places in the net. It considers the places as media processing objects and the transitions as the points of synchronization. Each place is associated with a duration, and all transitions occur instantaneously. The places rather than transitions have two different states: *active* and *inactive*, respectively indicated by *locked* and *unlocked* tokens.

An OCPN is a 6-tuple, $G_{OCPN} = (PL, TR, AR, MA, DU, RE)$ where, additionally,

$$DU : PL \rightarrow \mathbb{R} \qquad [\,durations]$$
$$RE : PL \rightarrow \{re_1, re_2, \ldots, re_k\} \text{ where } k \geqslant 0. \qquad [\,resources]$$

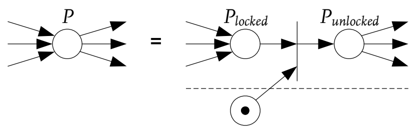
This is the original definition of OCPN as given by Little and Ghafoor (1990). Their firing rules are given in an informal way, which are summarized as follows:

1. When a token is added in a place, the place enters or remains in the active state and the token is *locked* for the duration specified or needed for media processing. When the place becomes inactive, or upon expiration of the duration, the token becomes *unlocked*.
2. A transition is enabled if all its input places contain an *unlocked* token.
3. Upon firing, the transition removes a token from each of its input places and adds a token to each of its output places.

One possible way to formalize this, is to let the marking $MA : PL \rightarrow \mathbb{N}$ be decomposed into a pair of markings $MA_{locked} : PL \rightarrow \mathbb{N}$ and $MA_{unlocked} : PL \rightarrow \mathbb{N}$ from which the original MA can be reconstructed as $MA(p) = MA_{locked}(p) + MA_{locked}(p)$ for $p \in PL$. An alternative would be to import the theory of colored Petri nets (Jensen, 1996). As far as known, Little and Ghafoor did not develop the formal definition of OCPN. Here a pragmatic approach is taken and continued with their practical definitions. Wherever possible it will be shown how the extensions, such as unlocking tokens, can be understood as "syntactic sugar"[4] for certain constructs in Petri nets.

If OCPN would be applied as the internal model for the synchronization engine, the timing specification in the following example IPML script can be converted into an OCPN as shown in figure 9.5 on the following page, where a play is cut into three sequential segments. In first two segments, a video stream, an audio stream and a robotic behavior need to be performed in parallel, assuming $\delta_{v_1} = \delta_{a_1} > \delta_{r_1}$ and $\delta_{v_2} = \delta_{a_2} > \delta_{r_2}$. In the last segment, the video stream v_3 and the robotic behavior r_3 are started at the same time in parallel, presumably $\delta_{v_3} > \delta_{r_3}$. According to these assumptions, although being separated into segments, the video streams v_1, v_2 and v_3 are expected to be played back continually without a break in between. The same holds for the audio streams a_1 and a_2.

[4]The locking extension to the Petri nets is considered as syntactic sugar: for each place P, two places P_{locked} and $P_{unlocked}$ are introduced with one transition T from P_{locked} to $P_{unlocked}$. All incoming transitions to P are incoming transitions to P_{locked} and all outgoing transitions of P depart from $P_{unlocked}$. The transition T can have one or more extra input places that belong to the player's or timer's presumed Petri net model that provides a "ready token" to enable T:



Syntactic sugar is a term coined by Peter J. Landin for additions to the syntax of a computer language that do not affect its expressiveness but make it "sweeter" for humans to use. For example C's "$a[i]$" notation is syntactic sugar for "$*(a + i)$".

```
<seq>
    <par>
        < action  id="v1" src="intro.mpg" />
        < action  id="a1" src="background1.au" />
        < action  id="r1" src="attention .bhv"  />
    </ par>
    <par>
        < action  id="v2" src="happystory.mpg" />
        < action  id="a2" src="background2.au" />
        < action  id="r2" src="happy_puppy.bhv" />
    </ par>
    <par>
        < action  id="v3" src="end.mpg" />
        < action  id="r3" src="quiet.bhv"  />
    </ par>
</ seq>
```

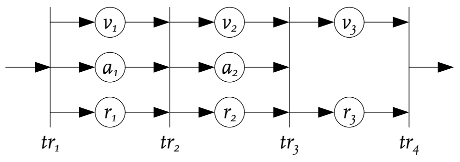


Figure 9.5: OCPN for an IPML example

However OCPN is not yet powerful enough to cover all the timing models in IPML. There are still a few problems that need to be solved, for example:

1. In figure 9.5, if a_1 and r_1 have finished performing but v_1 is late due the network overload and delay, transition tr_2 can not be enabled until v_1 finishes. Because of this delay, the audio stream a_2 can not be started even if its data has been ready. Audio streams are more jitter sensitive than video streams. There should be no perceivable break between the audio segments a_1 and a_2. Holding a_2 too long after finishing a_1 will result in audio jitter hence unacceptable audio quality. On the other hand, dropping a small portion of the video frames or having a small random delay between video frames may have a minor impact on the video quality that is perceivable by human eyes. In IPML the solution in handling this situation is to use the endsync to end the first par container when the audio stream a_1 finishes, no matter whether v_1 and r_1 have finished:

```
<par  endsync="a1">
    < action  id="v1" src="intro.mpg" />
    < action  id="a1" src="background1.au" />
    < action  id="r1" src="attention .bhv"  />
</ par>
```

The unfinished video frames in v_1 will be dropped. In OCPN a corresponding solution is to fire transition tr_2 right after a_1 is finished to maintain the audio quality regardless the token status of v_1 and r_1. However OCPN does not allow this type of transition – all the transitions can be enabled only after all its input places have an *unlocked* token.

2. OCPN needs to be extended to include places not only where the actors perform the actions, but also where the director controls the actions. For example the time values of the attributes of begin, end and dur can only be controlled outside the related action places because these values are related to containing timing containers or other action places. For example, the following IPML script requires the robotic behavior to be started 5 seconds after its containing par starts:

```
<par endsync="a1">
    <action id="v1" src="intro.mpg" />
    <action id="a1" src="background1.au" />
    <action id="r1" src="attention.bhv" begin="5s" />
</par>
```

An extra place that holds for 5 seconds before r_1 takes place is needed to complete the semantics above.

3. OCPN requires the actions taken in the places to have an explicit or implicit duration that should be determined when a place is activated. This results in difficulties in modeling nondeterministic durations and interaction events that are supported in IPML. As an example, values of begin and end attributes for the action places and the timing containers par and seq can be interaction events or other state change events from other actions or actors. How long these events will take to happen and whether these events will happen at all depends on the user interaction. The following change of the previous IPML example enables the robotic behavior r_1 to be started after 5 seconds, or whenever during the par element the user *activates* the video v_1 (for example, when the user clicks on the video presentation):

```
<par endsync="a1">
    <action id="v1" src="intro.mpg" />
    <action id="a1" src="background1.au" />
    <action id="r1" src="attention.bhv" begin="5s; v1.activateEvent" />
</par>
```

The OCPN model as is can not handle this type of nondeterministic behavior.

4. Some complicated synchronization features in IPML, such as restart, repeatDur and repeatCount, allow more flexible and dynamic temporal relations that depend on the runtime situations. It is not easy to produce the OCPN equivalences for these features. For example, if an element is required to repeat for repeatCount times, but within a total duration of repeatDur, the element in the OCPN structure may be duplicated for repeatCount times and these duplications may then concatenated together, each given a divided duration. However, if the repeatCount is a big number, this approach might result in too many duplications hence a huge representation.

Several researchers have attempted to improve the expressiveness of OCPN and address these problems. For example, Guan et al. (1998) propose Prioritized Petri Net (P-net) and an enhanced version (EP-net) (Guan and Lim, 2002) to add priority to the arcs in OCPN, and it can be used a solution for the endsync problem. However in an object-oriented implementation of OCPN, arcs are more naturally to be implemented as invocations and passing-message because they represent the movements of the process control. Further in IPML, the attribute endsync of a par container refers to one of its contained actions; it is more convenient to add priority to these actions when translating an IPML script into an OCPN representation. Yang and Huang (1996) propose a Real-time Synchronization Model (RTSM) based on OCPN and the idea of adding priorities to action places. These places are called "enforced places" – once an enforced place gets unlocked, the transition following it will be immediately fired regardless of the states of other places feeding this transition. This approach is proven powerful enough to cover most of the temporal constructs in SMIL (Yang, 2001; Yang, Tien, and Wang, 2003).

Based on the OCPN, and the aforementioned observations and research, the Action Synchronization Engine (ASE) is developed, which is to be presented next.

9.3.3 ASE

In short, the ASE is a runtime synchronization Engine that takes the timing and synchronization relations defined in an IPML script as input, and creates an object-oriented representation based on an extended version of OCPN.

An ASE model is a nine tuple that extends OCPN,

$$G_{ASE} = (PL, TR, AR, MA, DU, RE, PP, TC, CT),$$

where, additionally,

$$PP : \mathbb{P}\, PL \qquad [\,\textit{prority places}, \text{ a subset of } PL]$$
$$TC = \{tc_1, tc_2, \ldots, tc_q\} \text{ where } q > 0 \qquad [\,\text{Run-time } \textit{transition controllers}]$$
$$CT : \mathbb{P}(TR \times TC \times TR), \qquad [\,\text{TC controlled transition pairs}]$$

and where the functions DU and RE are redefined as follows,

$$DU : PL \rightarrow \{\mathsf{null}\} \cup \mathbb{R} \qquad [\,\textit{durations}]$$
$$RE : PL \rightarrow \{\mathsf{null}, re_1, re_2, \ldots, re_k\} \text{ where } k \geqslant 0. \qquad [\,\textit{resources}]$$

The inclusion of a null value in the ranges of the functions DU and RE means that there are places without a pre-determined duration, and there are places not related to any content resources.

The ASE model distinguishes *priority places* from other places. Special firing rules will be used for these priority places to implement the IPML endsync semantics and to cope with nondeterministic durations and interaction events. A priority place is drawn as a circle in an ASE graph like other places, but using a special (**thicker**) circle to emphasize its priority.

The added transition controllers TC make it possible to change the structure between two transitions in run time. It may fire another linked transition instead of the current enabled transition, which can be used to repeat or skip the structure between the controlled transition pairs. The controller may use a counter to control the number of repeat iterations, and may add and remove timer places in the structure to control the total duration for repeat. This mechanism is useful when dealing with IPML restart, repeatCount and repeatDur semantics. In an ASE graph, a box represents a transition controller, and dashed lines connect the controlled transition pairs (figure 9.6 on the facing page).

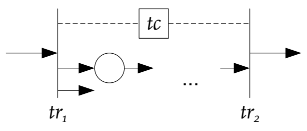


Figure 9.6: Transition controller

As already mentioned, there are places that do not have a pre-determined duration. The actual duration of these place can only be determined after the actions at these places have been carried out. These places are called *nondeterministic places*:
$NP = \{pl : PL \mid DU(pl) = \mathsf{null}\}$.
Non-deterministic places in an ASE graph are circles marked with a question mark.

Some nondeterministic places are not related to any content resources. These places are used in an ASE model to represent the actions that need to be taken by the engine itself to check certain conditions, to detect user interaction events, or simply to block the process etc. These places are called *auxiliary places*:
$AP = \{pl : NP \mid RE(pl) = \mathsf{null}\}$.

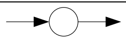
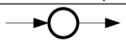
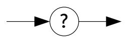
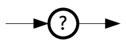
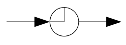
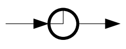
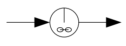
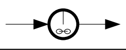
There are also places that do have a duration, but do not have a content element attached to it. These places are used by the ASE model to include an arbitrary interval to construct temporal relations. These places are called *timer places*:
$TP = \{pl : PL \mid DU(pl) \neq \mathsf{null} \wedge RE(pl) = \mathsf{null}\}$.

Timer places are indicated with a clock icon with the hands pointing to 9:00am. To construct an ASE graph from an IPML script structurally, it is sometimes necessary to connect two transitions. Since an arc can only be the link between a transition and a place, a zero-duration timer place can be inserted to maintain the consistency. These zero-duration timer places are called *connecting places*, indicated with a clock icon with its hands pointing to 0:00pm, and marked with an anchor link.

Table 9.3 on the next page shows the graph representations of the different ASE places and their priority versions. To show how an ASE would look like, the examples from section 9.3.2 on page 134 and on page 135 are combined below. The temporal structure in this example can be converted to the ASE model shown in figure 9.7 on the following page.

```
<seq>
    <par endsync="a1">
        <action id="v1" src="intro.mpg" />
        <action id="a1" src="background1.au" />
        <action id="r1" src="attention.bhv" begin="5s; v1.activateEvent" />
    </par>
    <par>
        <action id="v2" src="happystory.mpg" />
        <action id="a2" src="background2.au" />
        <action id="r2" src="happy_puppy.bhv" />
    </par>
    <par>
        <action id="v3" src="end.mpg" />
        <action id="r3" src="quiet.bhv" />
    </par>
</seq>
```

Table 9.3: Places in ASE

Place	Normal	Priority
Normal		
Nondeterministic		
Timer		
Connecting		

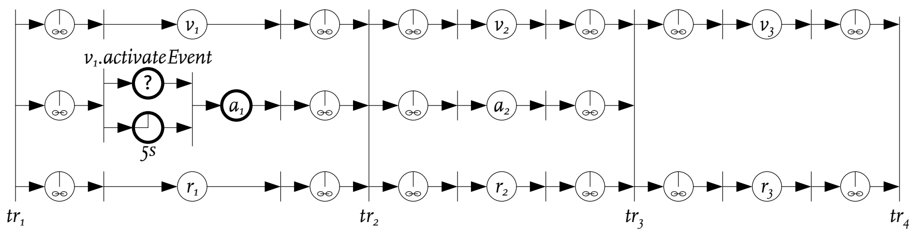


Figure 9.7: An example of the ASE model

An IPML script wraps all the timing elements in a `body` element. To compile the `<body> ... </body>` tag an "initial" place and a "final" place are added. The initial place contains precisely one unlocked token, whereas the final place has no token.

Detailed firing rules of ASE are included in background material N. Background material O describes how an IPML script can be translated to an ASE model.

9.3.4 Object-oriented implementation of ASE

The ASE in the IPML system is implemented in an object-oriented manner (figure 9.8 on the next page): Places and transitions are objects with input and output references that realize the arcs; transition enabling and firing are simply event-driven invocations. The Observer pattern can be used to implement the structure, where the transitions observe the token states of the connected places. Transition controllers are also objects with references to and from two related transitions. If the different types of the places are omitted from figure 9.8, the remaining static structure is rather simple. The dynamic behavior of these objects is driven by the firing rules and the implementation of the dynamic behavior is straightforward. The remaining design problem now is how to convert an IPML timing structure to an ASE model.

9.4 Get ready just in time

For an action to be immediately taken at the scheduled time, actors need to get ready prior to that time. For media objects enough amount of data needs to be prefetched; For robotic behaviors the mechanical system needs to be at a ready position for the next move. Two extreme strategies could be adopted by the director. First, the director

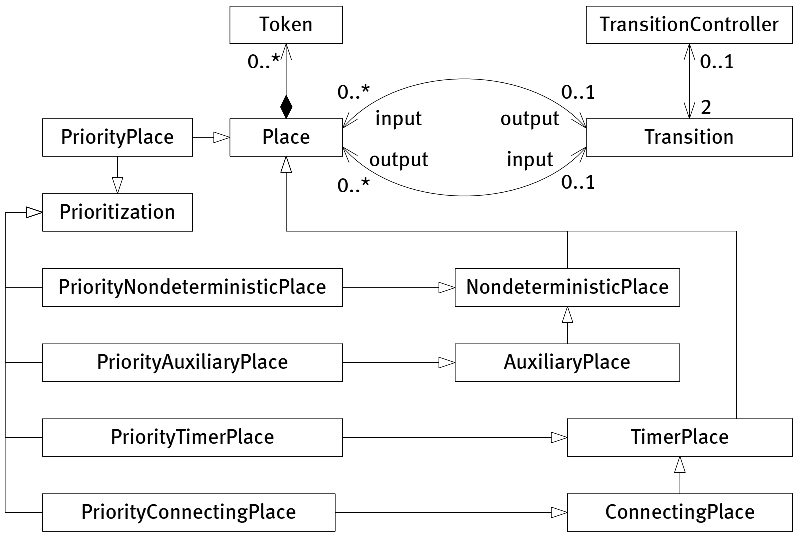


Figure 9.8: Object-oriented implementation of ASE

informs all actors to get prepared for all possible actions before the performance is started; second, the director never requests the actor to get ready before any action. The first strategy guarantees the smooth transitions between actions, and manages nondeterministic timing and user interaction well. However the cost is also obvious: it may result in a long initial delay and for media objects, and every actor needs a large buffer for prefetching all media objects in advance. The second strategy minimizes the initial start delay and the buffer requirement, but every transition between two actions will take time because the next action only starts to be prepared after the previous one stops. Hence smooth transitions between action places are not possible, unless the actions do not need to be prepared, which is rare in multimedia presentations. The nondeterministic user interactions make the situation even worse – Users may experience a long delay between their input actions and the system reactions. A different approach is needed for the IPML system.

9.4.1 Just-in-time approach

The director in the IPML system uses a "*just-in-time*" approach, in which the action preparation process is required to be completed just before the action time. With this strategy, the director informs the actor to prepare an action before the action time with the necessary preparation time taken into account. This strategy therefore requires less use of data buffers and facilitates more efficient use of network bandwidth.

In an ideal situation, i.e. the action time of all actions can be determined in advance, the *start-preparation* time for each action, that is, when an actor should start preparing an action, can be calculated based on its playback time, its QoS request, and the estimation of the available network bandwidth.

However for an IPML performance, the accurate action time often can not be determined before the performance takes place, because of the nondeterministic action durations and user interaction events. The best an ASE can do, is to predict the earliest action time for each action as if the nonmechanistic events would happen at the earliest possible moments. This can be done before the performance starts, as long as the ASE model has been established. During the performance, a dynamic error compensation mechanism can be used to adjust the estimate of the action time for each action and the start-reparation time as well.

9.4.2 Action time prediction

Once an ASE is converted and simplified from a given IPML, the director estimates the earliest possible action time for each action in the ASE by first assigning the duration of all nondeterministic places to zero and then traversing the ASE. The action time of an action is the firing time of its starting transition. There are two possible cases for a transition in the ASE: 1. it has no priority input place, or 2. it has at least one priority input place. For case 1, the firing time of the transition is the firing time of the preceding transition plus the maximum duration of the input places. For case 2, the firing time of the transition is instead the preceding transition plus the smaller one between the minimum duration of the priority places, and the maximum duration of the non priority places. Note that for time independent actions, such as presenting images and text, if the duration is not explicitly given, it is considered nondeterministic and its duration is considered as zero for prediction. For time-dependent actions, like presenting audio and video media objects, if the duration is not defined explicitly, the duration of the place is the implicit duration of the object if it can be determined from the server in advance.

During this traversal process of predicting the earliest possible firing time of each transition, it is also necessary to deal with the transition controllers to get more accurate values. The restart controller deals with events that could restart an element during the active duration of the element, so the earliest case for its ending transition would be that there is no restart at all. Thus, the restart controller is ignored during prediction. The repeat controller deals with the repeatDur attribute as well as the repeatCount attribute. The repeatDur attribute sets the duration of repeating an element, so the firing time of the ending transition should be the end of the repeat duration. The repeatCount attribute specifies the number of times to repeat, thus the firing time of the ending transition is extended as many times as specified. If any of them is is set to be "*indefinite*", the duration is considered nondeterministic hence a value of zero.

9.4.3 Dynamic adjustment

Obviously, the actual action time of every action will not be earlier than the prediction made prior to the performance. The differences between the actual action time of the actions that have already been taken and their predicted earliest times can be used to adjust the predicted action time of the actions that have not yet been performed. The predicted action time can then be updated for those yet to happen. The updated prediction of the action time can then be used to update the start-preparation time. Note that start-preparation time should not be updated if the preparation request has already been sent to the actor, since the actor may have already started preparation and an ongoing preparation process should not be interrupted. Nevertheless an ASE action time prediction with this dynamic adjustment mechanism does make the predicted action time of later actions closer to the actual action time, hence seems more intelligent than without.

9.5 Distributed time

Since they inhabit on different hardware platforms, actors may have time systems that are different from the directors. In order to get everything synchronized, the actors must use the director's time, or at least agree on the time difference. A simple approach to get actors have the same time as the director's, is to use clock synchronization mechanisms to synchronize the clocks of the underlying platforms.

9.5.1 Clock synchronization

An actor may perceive data skews due to asynchrony of its local clock with respect to the clock of the director, which may arise due to network delays and/or drifts in the clocks. In the absence of synchronized clocks, the time interval of an actor may have drifted to a value bigger or smaller than that of the director.

Clocks can be synchronized using an asynchronous protocol between the transport level entities in the presence of network delays compounded by clock drifts. Most clock synchronization protocols require the entities to asynchronously exchange their local clock values through the network and agree on a common clock value. These protocols use knowledge of the network delays in reaching agreement. For instance, the Network Time Protocol (NTP) requires the entities to receive their clock values from a central time server that maintains a highly stable and accurate clock and to correct the received clock values by offsetting the network delays. For clock synchronization protocols to function correctly, it is desirable that the network delay is deterministic, i.e., the degree of randomness in the delay is small and the average delay does not change significantly during execution of synchronization protocol. Accordingly, the transport protocol may create a deterministic channel with high loss and delay sensitivity to exchange clock control information. Clock synchronization is a complex topic of its own, and details of such protocols are outside the scope of this thesis.

9.5.2 Software clocks

The actors are not the only ones who inhabit a hardware platform. There may exist other hardware or software components sharing the same platform clock. Applying clock synchronization mechanisms to the shared clock may result in unexpected consequences on the components that are not under the supervision of the play director but have other time critical tasks of their own.

To avoid this side effect, the IPML system requires every actor to implement a software clock. The actor's clock is then synchronized with the director's clock using NTP according to the director's time on a regular interval basis. During the update interval, the actor's clock ticks ahead according to the local platform time.

9.5.3 "Action!" delayed

The director issues action scheduling commands over the network to the actors. "Action!" the director yells and expects the actor immediately starts the action. In real performance, these directing commands travel at the speed of sound and will be heard by the actor almost "immediately". However in the IPML system, these commands are not the only data traveling through the network. A command may need to be cut into pieces, packaged and queued at the director's side waiting for the network service to move it over. Once the packages arrive at the actor's side, they are again put in the queue for the network service to retrieve them. Once retrieved, depending on the network protocol, the data packages might need to be verified and confirmed before the command is reassembled and handed over to the actor. All these take time. Depending on the protocol and the bandwidth, it varies from few milliseconds to hundreds of milliseconds, or even more. The command will eventually be heard late. Since a particular network protocol is not assumed for transporting the commands, it is necessary to handle the delay at the architecture level.

Several strategies are adopted in the IPML system. First of all, all scheduling commands from the director are tagged with a time stamp that indicates when exactly the command is issued. Upon receipt, the actor retrieves the time stamp and compares it to its local software clock. Since the actor's clock is synchronized with the director's clock, the traveling time of the command can be calculated. If the command is not to start an action to present time-dependent content, the traveling time of the command is ignored. Otherwise if the traveling time is bigger than a QoS threshold, the actor will skip a fragment of time-dependent content that should have been performed right after the command was issued and before the command is received. Thus the distributed content elements can always be synchronized over the network, at the price of a small portion of the content being dropped at the beginning. If the network has enough bandwidth, the dropped content is hardly noticeable by the user.

9.6 Concluding remarks

This chapter presented a framework to translate an IPML script to a runtime scheduling engine, based on a formal approach that extends the Petri nets and OCPN. How timing can be managed in a distributed setting was also discussed based on this framework. Knowing that timing is a difficult issue, especially in a distributed setting, this chapter did not try to cover all possible issues. For example, when scheduling the content prefetching or preparation, the load of network bandwidth should be taken into account and be balanced. How this can be done is out of the scope of this thesis, but it is an important issue in a heavily loaded multimedia network.

CHAPTER 10

Mapping

Whereas formal specification and verification have shown their value by improving reliability and trustworthiness of traditional industrial systems, this chapter first presents a contribution that was made by applying them to the field of distributed theater play in an AmI context, to investigate the mapping problem in which content requirements are to be satisfied using given performance resources. Broy's stream-based component framework is used to model content-related interfaces and constraints in an elegant way. It combines the well-known notations of Z with an underlying concurrency theory. It shows that not only verification issues can be handled such as bandwidth and delay constraints, but also architecture-level issues such as network structural content-type compatibilities. The formalization is used as a theoretical framework and a starting point for the mapping problem in the IPML system: an IPML script describes the requirements on the performance cast by describing the actor types or the action content types, and these requirements need to be mapped to real configurations of available actors in a home theater to form a real performance cast, which may change during the action time because the actors may leave and join the theater at any time.

10.1 Mapping requirements to resources[1]

One of the problems of presenting multimedia to distributed AmI environments is the variety of such environments – think of how different people arrange their living rooms. AmI environments are not the only places to have such a problem. This has been a long-existing problem with regard to web browsers. In order to give the same look-and-feel on a single web page to the users, the poor authors often have to work very hard to use all kinds of tricks, and when this fails, write different versions for different browsers. On the other hand, web authors are lucky – if they only consider Internet Explorer, they will possibly cover $4/5$ of the audience, and if they

[1]This section is based on a paper published in the proceedings of The 28th Annual International Computer Software and Applications Conference (Feijs and Hu, 2004, COMPSAC 2004).

are kind enough to consider Mozilla Firefox users too, they will almost cover them all. However one can not assume that there are only Explorer-like living rooms or Mozilla-like living rooms. They are different, in terms of available components and connections. It is impossible to have one version to fit them all. Instead there should be only one abstract presentation script for all living rooms. The script should then be mapped to every room, to create the experience for end users as intended as possible. Similar mapping problems are studied informally by Kray, Krüger, and Endres (2003).

This chapter first aims at the formalization of the mapping from content requirements to devices. Each mapping problem is defined by a set of content requirements and a set of resources. The content requirements are abstract descriptions of the content element as well as its requirements for the physical components and their connections, including for example audiovisual players, robotic actors and lights. These physical components are presentation resources in a distributed presentation environment. Broy's stream-based component framework (Broy, 1999) is employed. The following issues should be addressed:

1. component interfaces for control and for events;
2. user interfaces: touch screens, switches and buttons;
3. throughput requirements and network delays;
4. embedding of presentation components into devices;
5. standardization of specifications.
   (keeping the math away from the content producers).

The formalization will then serve as a helpful theoretical framework and a starting point for the development of an automated mapper that can handle real action requirements and real performance resources (actors and channels) in an IPML play.

The mapping is to find the presentation resources for a set of content requirements so that the content element is properly presented. In a typical implementation a mapping is a process that at runtime deals with control and events, and that has access to a number of reserved communication and presentation resources, for example channels (chapter 7) and actors (chapter 8) in the IPML architecture. A distributed action is built by a local action service (also a factory, see section 6.3 on page 73) of the actor. This chapter adopts a pragmatic view, looking at the hardware resources such as presentation devices and network connections instead of software architectural abstractions such as actors and channels.

10.1.1 Preliminaries

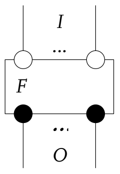
To get started a few simple content requirements are considered:

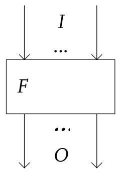
- high-resolution video presentation
- low-resolution video presentation
- dancing behavior
- up-down interactor

which have to be obtained from the following resources:

- *cable*: upc cable 801(10Mb/s)
- *the_sign*: www.clips.com/the_sign.mpg (10Kb/s)
- *happy_puppy*: www.dance.com/happy_puppy.bhv (1Kb/s)
- *updown*: /interactors/updown.exe (1b/s).

To model the content and connection requirements, the approach of Broy (Broy, 1999) based on timed streams is followed. Some additional drawing conventions are used, which is derived from ADL Darwin (Eixelsberger and Gall, 1998; Feijs and Qian, 2002), and the object oriented view on channels (see figure 7.10 on page 98).



is used instead of


(Broy, 1999, p.7).

As discussed in chapter 8, in an interactive system, it is necessary to distinguish user interface from streaming channels. For this the convention introduced in chapter 8 on page 104 is used, that is, output channels ■ modeling real, physical presentations such as sounds, video frames and robotic movements, and input channels □ modeling the components with which the user may interact with the system, such as a GUI interface on a screen and physical buttons on a remote control.

Broy demands that a set S of types be given. Let's put

$$S = \{PAL, MPG, DBH, IBH, \mathbb{R}\}$$

for PAL streams, MPEG streams, Dancing BeHavior, Interaction BeHavior and "things in the ℝeal world"(for example, buttons, screens and robots).

In Broy's approach there is a discrete time frame representing time as an infinite chain of time intervals of equal finite duration. Let's take 1 second for that duration. This allows the specification to formally represent the facts for example that there are 25 video frames per second in a high-resolution video stream and that each frame takes a number of bits.

Let's assume the functions on data:

$$\begin{array}{rll} bits: & PAL \to \{0,1\}^* & \text{... also for } MPG, DBH \\ pr: & PAL \to \mathbb{R} & \\ pr: & MPG \to \mathbb{R} & \\ pr: & DBH \to \mathbb{R} & \\ pr: & IBH \to \mathbb{R} & (pr \text{ for presentation}). \end{array}$$

and lifted versions of pr are abtained at timed-stream level:

$$\begin{array}{rll} pr': & PAL^* \to \mathbb{R}^* & \text{by } (pr'(p)).i = pr(p.i) \\ pr'': & (PAL^*)^\infty \to (\mathbb{R}^*)^\infty & \text{by } (pr''(x)).t = pr'(x.t) \end{array}$$

and so on. The formal model has some structure of its own (figure 10.1).

In order to map this, devices and connections must be assumed. Figure 10.2 on the next page shows a possible configuration with the following devices:

- *STB*: set-top box;
- *modem*: internet modem;
- *hi_res&PIP*: high-resolution monitor with picture-in-picture function;
- *lo_res*: low-resolution (lo_res) monitor;
- *level2_robot*: "level-2" robot;
- *RC*: remote control for the low-resolution monitor

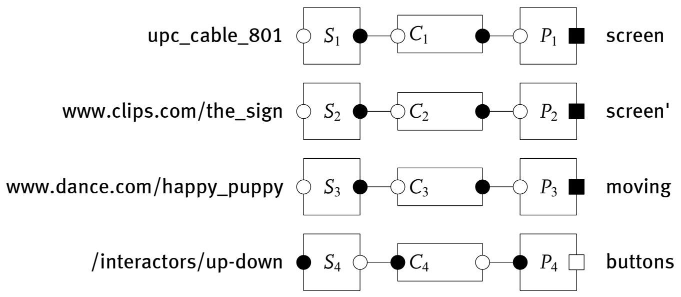


Figure 10.1: Formal model structure

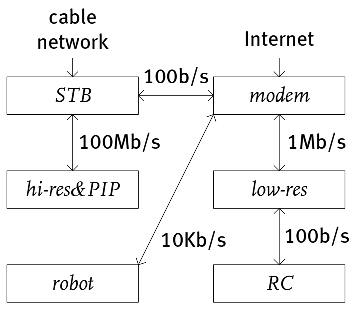


Figure 10.2: A possible configuration

and the following connections:

- *STB—modem* 100b/s duplex;
- *STB—hi_res* 100Mb/s duplex;
- *modem—lo_res* 1Mb/s duplex;
- *modem—robot* 10Kb/s duplex;
- *RC—lo_res* 100b/s duplex.

There are two possible ways of presenting the dancing behavior: either by an animated character on the *PIP* of the *hi_res* screen, or by the physical robot together with the *RC* providing user input. But the *STB—modem* bandwidth is not enough for the *PIP* to do it. So the robot must dance and the *RC* must provide the up-down interaction.

10.1.2 Specification

Some abbreviations are needed. For timed stream z and $n_1, n_2 \in \mathbb{N}$,

$$\begin{aligned} rate(z, n_1, n_2) &== \forall\, t : \mathbb{N}\backslash\{0\} \bullet \#(z.t) = n_1 \\ &\quad \wedge \forall\, i : \mathbb{N} \bullet 0 < i \leqslant \#(z.t) \Rightarrow \#bits(z.t.i) = n_2 \end{aligned}$$

with the intuition that e.g. $rate(z, 100, 100\text{K})$ means that 100 frames fit into once second and that 100K bits go into each frame (figure 10.3 on the facing page).

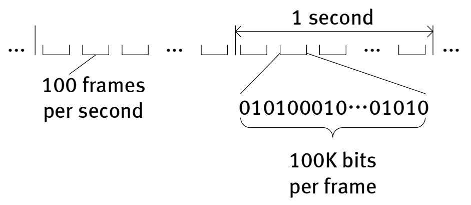


Figure 10.3: $rate(z, 100, 100K)$

For timed streams x and x', and $\Delta \in \mathbb{N}$,

$$delay(x, x', \Delta) == \forall t : \mathbb{N} \backslash \{0\} \bullet x.t = x'.(t + \Delta)$$

and for $m \in \mathbb{N}$,

$$maxdelay(x, x', m) == \exists \Delta : \mathbb{N} \bullet \Delta < m \wedge delay(x, x', \Delta).$$

The rate information is put into the source devices S_1, S_2, S_3 and the sink S_4.

S_1
in $cable : PAL$
out $z : PAL$

$z = cable$
$rate(z, 100, 100K)$

S_2
in $the\_sign : MPG$
out $z : MPG$

$z = the\_sign$
$rate(z, 2, 5K)$

S_3
in $happy\_puppy :$
DBH
out $z : DBH$

$z = happy\_puppy$
$rate(z, 10, 100)$

S_4
in $z : IBH$
out $updown : IBH$

$z = updown$
$rate(z, 1, 1)$

The relation between media formats and the real world is described in $P_1, ..., P_4$.

P_1
in $w : PAL$
out $screen : \mathbb{R}$

$screen = pr''(w)$

P_2
in $w : MPG$
out $screen' : \mathbb{R}$

$screen' = pr''(w)$

P_3
in $w : DBH$
out $moving : \mathbb{R}$

$moving = pr''(w)$

P_4
in $button : \mathbb{R}$
out $w : IBH$

$button = pr''(w)$

Matters of delay are modeled in the channels.

C_1
in $z : PAL$
out $w : PAL$

$maxdelay(z, w, 2)$

C_2
in $z : MPG$
out $w : MPG$

$maxdelay(z, w, 3)$

C_3
in $z : DBH$
out $w : DBH$

$maxdelay(z, w, 4)$

C_4
in $z : IBH$
out $w : IBH$

$maxdelay(z, w, 4)$

Finally the whole system is specified by:

$$\begin{aligned} SYSTEM = {} & (S_1 \circ C_1 \circ P_1) \setminus \{z, w\} \\ & \parallel (S_2 \circ C_2 \circ P_2) \setminus \{z, w\} \\ & \parallel (S_3 \circ C_3 \circ P_3) \setminus \{z, w\} \\ & \parallel (P_4 \circ C_4 \circ S_4) \setminus \{z, w\} \end{aligned}$$

which has a syntactic interface of (I, O), where

$$\begin{aligned} I &= \{cable, the\_sign, happy\_puppy, button\} \\ O &= \{screen, screen', moving, updown\}. \end{aligned}$$

10.1.3 Presentation resources

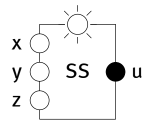
The next step is to formalize the presentation resources, i.e. the components and the network connections. Let's assume that the components cause delays and that the connections give rise to bandwidth restrictions. The structure of the resulting model is shown in figure 10.4 on the next page. Clearly renaming will be needed later, for example, $[the\_sign/url_1]$ (for modem) and $[cable/channel]$ (for STB).

Somehow the components must be configured to perform the required routing of streams. Although the current mapping-example does not require it, in general a component must be able to combine two incoming streams and put them on a single output port, or conversely, split what comes in to produce separate outgoing streams. In other words, some components must contain a switch.

A component containing a switch must have an extra channel of a special type, a control channel, accepting so-called *commands*, from a set $\mathbb{C}$. So from now on,

$$S = \{PAL, MPG, DBH, IBH, \mathbb{R}, \mathbb{C}\}.$$

To get some experience in modeling switches, let's try a simplified component, the Simple Switch SS:



$(x, y, z : DBH)$

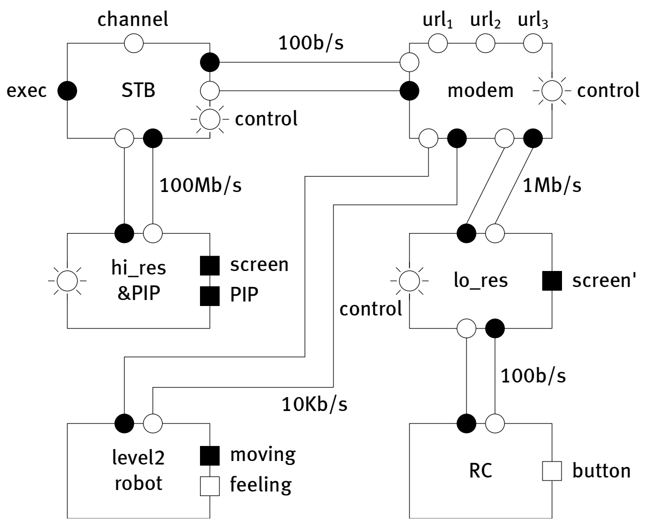


Figure 10.4: Structure of the resource model

Any selection from x, y and z can be combined and offered on u, provided the rates fit. To adapt to the rate of u (assumed to be fixed), dummy data must be stuffed.

Let's assume the existence of disjoint copies of the set DBH, which are denoted as DBH_1, DBH_2, etc. Let's also assume conversion functions $d_1 : DBH \rightarrow DBH_1$, $d_2 : DBH \rightarrow DBH_2$, etc. and the corresponding inverses, e.g. $d_1^{-1} : DBH_1 \rightarrow DBH$. They give rise to lifted versions, e.g. $d_1' : DBH^* \rightarrow DBH_1^*$. These assumptions allow the specification to describe streams being merged, e.g. x and y merged into u:

$$u \copyright DBH_1 = d_1''(x) \wedge u \copyright DBH_2 = d_2''(y).$$

In other words, the mechanism of disjoint copies is used to model tagging of sub-streams conveniently in an abstract way. (In Broy's theory, $S \copyright x$ means the stream obtained from x by deleting all its messages that are not elements of the set S.)

Let's introduce abbreviations:

$$maxrate(x,n) == \forall\, t : \mathbb{N}\backslash\{0\} \bullet \sum_{i=1}^{\#(x.t)} \#bits(x.t.i) < n$$
$$maxrate(x,y,n) == \forall\, t : \mathbb{N}\backslash\{0\} \bullet \sum_{i=1}^{\#(x.t)} \#bits(x.t.i) + \sum_{i=1}^{\#(y.t)} \#bits(y.t.i) < n$$

Although a real network connection carries bits, a more abstract model to reflect the general-purpose nature of connections can be used:

$$\mathbb{B} = (\bigcup_{i=0,1,2\ldots} DBH_i) \cup (\bigcup_{i=0,1,2\ldots} PAL_i) \cup \text{etc.} \qquad \text{(all data types used)}$$

where $DBH_0 \triangleq DBH$ (So this can be used for the simple case where tagging is not really needed).

SS

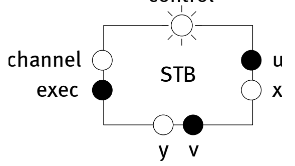
in $x, y, z : DBH$
in $s : \mathbb{C}$
out $u : \mathbb{B}$

$s.1.1 = xy2u \Rightarrow$
$\quad maxrate(x, y, 100)$
$\quad \wedge\ delay(d_1'(x), u\copyright DBH_1, \Delta_{SS}) \wedge delay(d_2'(x), u\copyright DBH_2, \Delta_{SS})$
$s.1.1 = yx2u \Rightarrow$
$\quad maxrate(x, y, 100)$
$\quad \wedge\ delay(d_2'(x), u\copyright DBH_2, \Delta_{SS}) \wedge delay(d_1'(x), u\copyright DBH_1, \Delta_{SS})$
$s.1.1 = x2u \Rightarrow$
$\quad maxrate(x, 100) \wedge delay(d_1'(x), u\copyright DBH_1, \Delta_{SS})$
etc. $(y2u, z2u, xz2u, \ldots)$

Based this example, let's consider the real components, assuming the real delay occurs in the components as show in the following table. This means that a constraint such as $delay(x, u, \Delta_{SS})$ has to be rewritten as $delay(d_1''(x),\ u\copyright DBH_1,\ \Delta_{SS})$. Let's assume the delay is fixed, independent of the load and the precise routing. Otherwise, more sophisticated schemes can be devised if necessary. Note that $\copyright'$ is the lifted version of $\copyright$. The rate constraints are modeled as if they belong to the input ports.

component	delay	value
STB	Δ_{STB}	1
modem	Δ_{modem}	1
hi_res	Δ_{hi_res}	1
lo_res	Δ_{lo_res}	1
level2_robot	Δ_{level2_robot}	0
RC	Δ_{RC}	0

The presentation resources are then specified as follows.

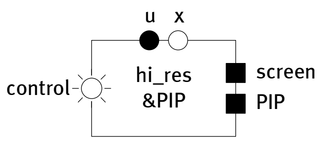
STB :


STB

in $channel : PAL$
in $control : \mathbb{C}$
in $x, y : \mathbb{B}$
out $exec : IBH$
out $u, v : \mathbb{B}$

$control.1.1 = channel2v \Rightarrow$
$\quad maxrate(channel, \text{100M}) \wedge delay(channel, v\copyright PAL, \Delta_{STB})$
$control.1.1 = x2exec \Rightarrow$
$\quad maxrate(x, \text{100K}) \wedge delay(x\copyright DBH, exec, \Delta_{STB})$
$control.1.1 = channel2v\_x2exec \Rightarrow$
$\quad maxrate(channel, \text{100M}) \wedge maxrate(x, \text{100K})$
$\quad \wedge\ delay(channel, v\copyright PAL, \Delta_{STB})$
$\quad \wedge\ delay(x\copyright DBH, exec, \Delta_{STB})$
etc.

hi_res&PIP:

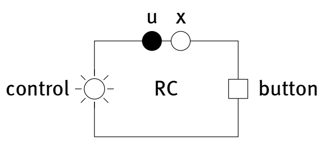


hi_res&PIP

in $x : \mathbb{B}$
in $control : \mathbb{C}$
out $u : \mathbb{B}$
out $screen, PIP : \mathbb{R}$

$control.1.1 = x2screen \Rightarrow$
$\quad maxrate(x, \text{100M})$
$\quad \wedge\ delay(pr''(x\copyright PAL), screen, \Delta_{hi\_res\&PIP})$
$control.1.1 = x2screen\_PIP \Rightarrow$
$\quad maxrate(x, \text{100M})$
$\quad \wedge\ delay(pr''(x\copyright PAL), screen, \Delta_{hi\_res\&PIP})$
$\quad \wedge\ delay(pr''(x\copyright DBH), PIP, \Delta_{hi\_res\&PIP})$
etc.

RC:

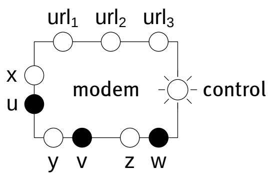


RC

in $x : \mathbb{B}$
in $control : \mathbb{C}$
in $button : \mathbb{R}$
out $u : \mathbb{B}$

$control.1.1 = button2u \Rightarrow$
$\quad maxrate(u, 100) \wedge delay(button, pr''(u\copyright IBH), \Delta_{RC})$

modem:

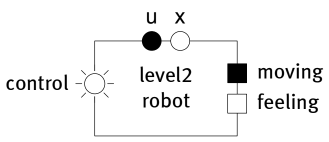


modem

in $url_1, url_2, url_3, x, y, z : \mathbb{B}$
in $control : \mathbb{C}$
out $u, v, w : \mathbb{B}$

$control.1.1 = url_1 2v \Rightarrow$
$\quad maxrate(url_1, 56K) \wedge delay(url_1, v, \Delta_{modem})$
$control.1.1 = z2u \Rightarrow$
$\quad maxrate(z, 1M) \wedge delay(z, u, \Delta_{modem})$
$control.1.1 = url_2 2w \Rightarrow$
$\quad maxrate(url_2, 56K) \wedge delay(url_2, w, \Delta_{modem})$
etc.

level2_robot:

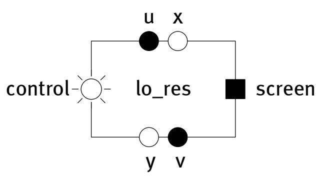


level2_robot

in $x : \mathbb{B}$
in $feeling : \mathbb{R}$
in $control : \mathbb{C}$
out $moving : \mathbb{R}$
out $u : \mathbb{B}$

$control.1.1 = x2moving \Rightarrow$
$\quad maxrate(x, 10K) \wedge delay(pr''(x \circledcirc DBH), moving, \Delta_{level2\_robot})$
$control.1.1 = feeling2u \Rightarrow$
$\quad maxrate(u, 10K) \wedge delay(feeling, pr''(u \circledcirc IBH), \Delta_{level2\_robot})$
$control.1.1 = x2moving\_feeling2u \Rightarrow$
$\quad maxrate(x, 10K) \wedge maxrate(u, 10K)$
$\quad \wedge\ delay(pr''(x \circledcirc DBH), moving, \Delta_{level2\_robot})$
$\quad \wedge\ delay(feeling, pr''(u \circledcirc IBH), \Delta_{level2\_robot})$

lo_res:



lo_res

in $x, y : \mathbb{B}$
in $control : \mathbb{C}$
out $u, v : \mathbb{B}$
out $screen : \mathbb{R}$

$$\begin{array}{l}
control.1.1 = x2screen \Rightarrow \\
\quad maxrate(x, \mathsf{1M}) \wedge delay(pr''(x \copyright MPG), screen, \Delta_{lo\_res}) \\
control.1.1 = y2u \Rightarrow \\
\quad maxrate(y, 100) \wedge delay(y, u, \Delta_{lo\_res}) \\
control.1.1 = x2screen\_y2u \Rightarrow \\
\quad maxrate(x, \mathsf{1M}) \wedge maxrate(y, 100) \\
\quad \wedge\ delay(pr''(x \copyright MPG), screen, \Delta_{lo\_res}) \\
\quad \wedge\ delay(y, u, \Delta_{lo\_res})
\end{array}$$

etc.

10.1.4 Mapping

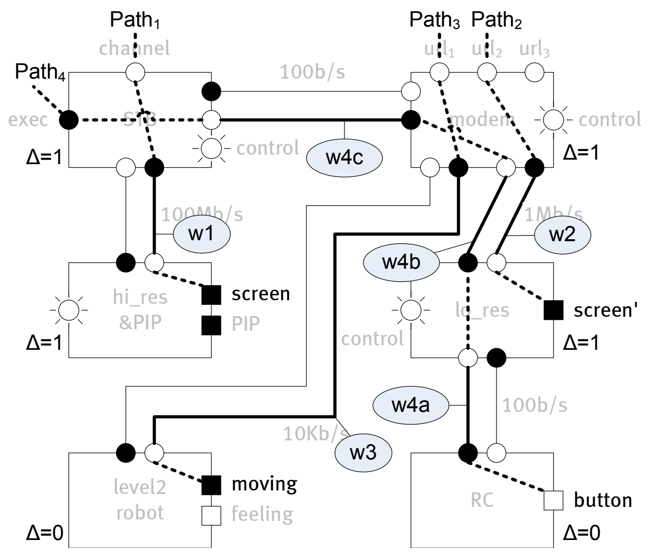
Next a mapping is proposed. Abstractly, a mapping is a set of paths through a given network, one path for each of the four chains specified in *SYSTEM*. At a concrete level, the paths are established by feeding appropriate switching commands into *STB*, *modem* and *lo_res*. In general there is a freedom in choosing these paths, for example if alternative routing through the network exists. Even the choice of which screen or interactive input is to be used is not a priori fixed. The four paths are shown in figure 10.5 on the following page. In an informal way it is easy to check that the delay and bandwidth constraints are not violated and the mapping is feasible.

To formalize the connections renaming is used, adapting port names to "wire names". The wire names are also shown in figure 10.5 on the next page. Note that *w4a*, *w4b* and *w4c* together form a path from *RC* via *lo_res* and *modem* to *STB*. The other wire names are not used. Unused ports must be hidden and ports that go to the external world must be renamed.

$$\begin{array}{l}
STB' = STB[cable/channel, updown/exec, w4c/x, w1/v] \\
\quad \ll control : channel2v\_x2exec \setminus (control, x, y) \\
hi\_res\&PIP' = hi\_res\&PIP[w1/x] \ll control : x2screen \setminus (control, u, PIP) \\
RC' = RC[w4a/u] \ll control : button2u \setminus (control, x) \\
lo\_res' = lo\_res[w4a/y, w4b/u, w2/x, screen'/screen] \\
\quad \ll control : x2screen\_y2u \setminus (control, v) \\
modem' = modem[happy\_puppy/url_1, the\_sign/url_2, w4c/u, w3/v, w4b/z, w2/w] \\
\quad \ll control : url_1 2v\_url_2 2w\_z2u \setminus (control, url_3) \\
level2\_robot' = level2\_robot[w3/x] \ll control : x2moving \setminus (control, u, feeling)
\end{array}$$

Finally the whole system implementation is described by

$$SYSTEM' = (STB' \otimes hi\_res\&PIP' \otimes RC' \otimes lo\_res' \otimes modem' \otimes level2\_robot')$$



	Media needs	$\sum\Delta$	Required $\sum\Delta$
path$_1$	cable	2	$\leqslant 2$
path$_2$	the_sign	1	$\leqslant 3$
path$_3$	happy_puppy	1	$\leqslant 4$
path$_4$	updown	3	$\leqslant 4$

Figure 10.5: Four paths and "wire names"

which has the syntactic interface (I', O'), where

$$I' = \{cable, the\_sign, happy\_puppy, button\}$$
$$O' = \{screen, screen', moving, updown\}.$$

The next questions is: how to express the formal correctness of the implementation? Background material P includes a correctness proof of the proposed mapping.

10.1.5 Discussion

The case study shows how Broy's framework can deal with special controls and events at an abstract level and with real-world interfaces. It can also deal with bandwidth requirements and network delays very well. The case study does not have a sophisticated performance model, but it is plausible that certain models can be made using the same modeling style (for example if the delay is a function of the bit rate).

The case study shows one specification and one configuration of presentation resources. Abstractly, the mapping between the two is a set of paths through the network. Concretely, a mapping is a set of control commands, to be given to those components that have switching capabilities. Maximal rate and delay requirements can be checked formally, although the calculations are not surprising. Media types are described by sets such as *PAL* and *MPG*, this implies that type compatibilities are handled formally too.

The model created focuses on the stream-based aspects; in this type of distributed systems these aspects are most important at the architectural level. At a protocol level, it is expected that reactive behavior is most important, and complementary state-machine based models could be used (Ulrich and König, 1997).

The following research questions remain as options for future research: 1. How to model dynamic aspects such as changing configurations and servers that build presentations according to the "factory" paradigm; 2. How to scale-up the approach when more and more technicalities have to be modeled (while keeping the math away from media developers); 3. How to specify the general class of "mapping problems" (instead of a single instance, as it was done now).

10.2 IPML mapping

As learnt from the formal study on the mapping issues in the previous section, a mapping process is a set of controlling commands, to be sent through control channels to the components that are capable of copying and combining streams from input channels to output channels. All the actors in the IPML system are all connected through channels in a PAC hierarchy and controlling commands can be sent though these connecting channels (see chapter 8). Incorporating the *Simple Switch* mechanism in the actors for copying and combining streams is also straightforward. This section first show how this can be done, then briefly presents the dynamic mapping in IPML system, which was not yet covered by the formal study in the previous section.

10.2.1 Implementing the Simple Switch

Since the Channel patterns (see chapter 7 on page 77) are applied in the IPML system to implement the communication channels, the switch mechanism proposed in the previous section can be easily implemented. Let's take the Streaming Channel pattern as an example. Suppose an actor needs to combine input streams and send the combined data stream to a output channel *c* through a proxy (*ProxyStreamConsumer*) *psc*, the actor may use a output (*Streamsupplier*) *u* to connect to *c*. The output port *u* can be implemented as an object of the class *SimpleSwitch* as follows:

SimpleSwitch

$\upharpoonright(Push, DisconnectStreamConsumer, DisconnectStreamSupplier)$

$SteamConsumer, StreamSupplier$

$p : ProxyStreamConsumer$

$Push \mathrel{\widehat=} p.Push$

$DisconnectStreamConsumer \mathrel{\widehat=} [\ ]$

$DisconnectStreamSupplier \mathrel{\widehat=} [\ ]$

where $u.p = psc$. Note that a *SimpleSwitch* object can act as not only an output port (*StreamSupplier*), but at the same time an input port (*StreamConsumer*). If two input streams from two channels need to be combined, the actor can simply get a *ProxyPushSupplier* from each of them and connect *u* to the proxy suppliers to receive the data. If any of the input stream is no longer needed to be copied or combined, the actor may disconnect *u* from the input channels at any time.

An alternative approach to implement the switch is to use channel compositions to directly connect and disconnect the proxy suppliers from the input channels and the proxy consumers from the output channel without using an intermediate port, as explained in section 7.2.4 on page 84 (also see figure 7.6(a) on page 86). However, with the intermediate port, other mechanisms, for example filtering, can be implemented inside this port as well, if necessary.

Now all the actors are capable of doing what is required for mapping: listening to the controlling commands, connecting and disconnecting channels to copy, combine and cut off the streams, as it is specified in the previous section.

10.2.2 Dynamic Mapping

Chapter 4 presented how the requirements on the performance cast can be specified by describing the actor types in an IPML script. Chapter 8 on page 99 formally specified the architecture of the IPML system where the real actors are connected to the director via the virtual actors through channels (chapter 7) in a PAC hierarchy. When all these come together, dynamic mapping is not anymore a mission impossible, but rather easy instead. Here how the mapping can be handled in a dynamic setting is briefly described.

Virtual actors

Virtual actors are required in the IPML system architecture (figure 8.7 on page 112) as an essential layer of software PAC agents for dynamic mapping. These agents can be provided by the vendors of the real actors as a software driver, or by the content producers as an "recommended" actor if there is no real actor available. These virtual actors can be provided by an installation package which requires the user to install it in advance, or for example an Internet resource identified by a Uniform Resource Locator (URL) such that the virtual actor can be downloaded and installed automatically. Here one shall not try to cover the security and privacy consequences of this automatic downloading and installation process, since it has been an issue for all Internet applications and should be taken care of by dedicated protocols and subsystems.

Once the virtual actors are available to the IPML system, it is then registered and maintained by the mapping engine of the director.

Channel resources

The system also provides and maintains a distributed channel service over the connected devices. Here it benefits from the design of the channel patterns presented in chapter 7: all channels between actors and the director are distributed objects managed by a channel service, hence the network resources can be easily monitored and allocated with QoS and load balancing taken into account. The director may query the channel service so that the communication conditions can be taken into account during the mapping process. The example in the previous section can be used again – the description of "But the *STB–modem* bandwidth is not enough ... So the robot must dance ..." on page 146 now has a concrete footnote on how this can be reasoned about in the IPML system.

Mapping heuristics

The IPML director has a list of available virtual actors together with their types given. The director also has access to the channel service to query the channel resources to find out whether a virtual actor is connected to a real actor. Given an actor type as the requirement, the IPML uses the following heuristics to map the required actor type to a virtual actor:

1. The user preference has top priority (see the "naughty boy" example on page 47).
2. If after 1 multiple virtual actors can be selected, the ones having the"closest" type have the priority over the others.
3. If after 2 multiple virtual actors can be selected, the ones with a real actor connection have priority over those without.
4. If after 3 multiple virtual actors can be selected, the one that has been selected most recently for this type is again selected. If none of them have ever been selected, the director randomly selects one from these virtual actors.
5. If none of the virtual actors can be selected, the director creates a "dummy" virtual actor for this type. The "dummy" virtual actor will do nothing but ignore all requests.

In step 2, how to decide an actor is the "closest" to another among the others is not clearly described. It depends on how the types are defined. For example it is possible to apply a method that is similar to the one used for determining the "closest" media type in section G.3 of background material G. In practice, one may also leave it to an ontology reasoning system for example a semantic web tool for RDF or OWL type descriptions, as it was briefly described in section 4.3.1 on page 46. Here let's leave it open to implementation.

During the action time, these heuristic conditions may change, for example, the real actors may connect and disconnect from the "theater" at any time, and users may change their minds at any time to have a "gentleman" instead of a "naughty boy" to be the actor or vice versa. To dynamically update the mapping relations, the director needs to repeat this mapping process on a regular interval basis.

Actor/director discovery

The problem now is how the virtual actors, the real actors and the director can find each other for registration and connection. This is actually a well-known device/service discovery problem and many middleware standards table 10.1 on the following page have a solution for it. For example, JINI and Home Audio Video Interoperability (HAVi) offer software elements like central *lookup service* and distributed *registry* respectively, where the devices and services register and make themselves "visible" to the other devices and services on the network. The clients query the lookup service or the registry via APIs in order to find the required service or device. UPnP and Video Electronic Standards Association (VESA) offer a protocol based solution for device/service discovery. UPnP uses the Simple Service Discovery Protocol (SSDP) for device discovery, whereas VESA provides a software agent called Home Network Broker (HNB) which helps the devices to find each other. In Open Services Gateway Initiative (OSGi), the device discovery is done by OSGi *device access system* which supports automatic attachment and detachment of devices and can automatically download and start appropriate device drivers; devices can be plugged and unplugged at any time and the device access system immediately responds to changes.

Table 10.1: Middleware standards

Standard	Data transport	Developed by	Information
JINI	IP over any media	Sun Microsystems	www.jini.org
HAVi	FireWire	Philips, Sony, Hitachi, etc.	www.havi.org
UPnP	IP over any media	Microsoft	www.upnp.org
VESA	FireWire and IP	Samsung, Canon, HP, etc.	www.vesa.org
OSGi	Any media	Ericsson, IBM, Philips, etc.	www.osgi.org

So one may simply leave the discovery task of registering virtual actors to the director and the task of connecting virtual actors and the real actors to these middleware infrastructures. In the experimental implementation, JINI was used wherever it is supported in the flavor of standard Java technology. The OSGi service was also used for connecting a User Interface Markup Language (UIML) based GUI actor on Philips iPronto because of its native OSGi support, also for experimenting with a different discovery service. But the structure proposed here does not rely on a particular middleware standard.

10.3 concluding remarks

This chapter did not try to fully formalize the dynamic mapping in distributed environments, however, the exercise in formally specifying the mapping with a case study helped the project gain enough insights in general mapping problems and solutions. The work that had been done in formalizing the architectural components such as actions, channels and actors, also helped turn these insights easily into practice when the IPML mapping engine was implemented. These formal specifications contributed to the design process by clarifying the idea and the concepts, locating the potential design problems and bringing up the solutions at a high level of abstraction other than loosing the focus in low level coding.

However, without coding, an ingenious formal speciation can never run and sing. The best way to test these designs is to implement them, which is exactly the topic coming up next.

Part IV

Evaluation

CHAPTER 11

Implementations

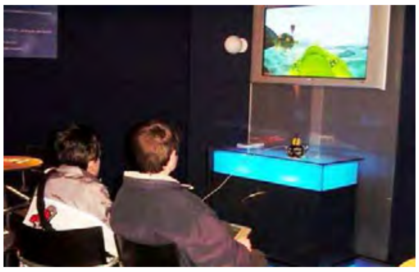
As mentioned early in the first chapter, this project went through three design iterations. During the first iteration, StoryML was developed and a demonstrator called TOONS was implemented, as it is described in part I. During the second and the third iteration, the concepts of the IPML design described in part II and part III were gradually polished and put into practice. This chapter briefly presents several demonstrators or applications that were implemented based on the IPML architectural design.

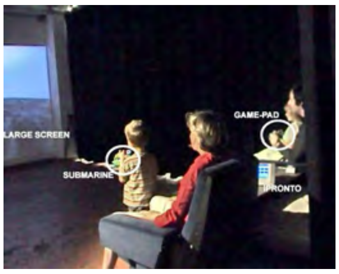
11.1 DeepSea (2001-2003)

The DeepSea application, as a result of the ICE-CREAM project, is a distributed interactive movie developed by Philips Research and de Pinxi (2003). Part of this PhD project, supported by Philips Research, was carried out during the ICE-CREAM project by contributing to DeepSea with the architectural design and participating in the implementation to carry out the design concepts in the practice. De Pinxi as an interactive 3D movie provider has a lot of experience in creating interactive experiences in the domains both of leisure and of education with the aim of "immersing" the participants into a truly interactive context (de Pinxi, 2003). De Pinxi not only contributed to the project with the content and a proprietary 3D movie engine to render the content, but also brought in their expertise in interactive and immersive theaters to create impressive lighting and sound effects. Figure 11.1(a) on the next page shows a demonstration setting exhibited in IBC (2003), and figure 11.1(b) on the following page shows a family experiencing DeepSea.

11.1.1 Content

The final content structure follows the storyboard developed by de Pinxi early at the beginning of the project (see section 2.2.2 on page 22 for details). The movie presents an underwater virtual space for the user to explore. Figure 11.1(a) on the following page shows the content structure of the movie. The movie starts with a submarine


(a) DeepSea exhibited at IBC 2003



(b) A family experiencing DeepSea

Figure 11.1: DeepSea Demonstrations

sailing on a sea and after a while diving into the deep sea. When the submarine reach the sea bed, the user will find a small TV set with a message that helps the user to customize the presentation. The user may select a presentation language (English, Dutch or French) and a presentation mode. There are three presentation modes:

Automated mode: This is also the default mode if the user does not select a mode within a time limit. In this mode, the presentation becomes a traditional TV program that presents the underwater scenery. The submarine moves slowly in the space following a scripted route. The user will see different kinds of fish, weeds, shipwrecks and landscapes.

Game mode: This mode allows the user to move the submarine in two directions, left and right, to navigate the 3D space following the scripted route. The submarine drives ahead in a fixed speed and the user does not have all the freedom to move the submarine around. The user will get a score reward by avoiding obstacles and mines in the route by maneuvering the submarine to the left or right. When a mine is hit, the movie shows the explosion with surrounding sounds, lighting effects and vibration effects. The user will get a final score that correlates to the performance of avoiding the mines and the time used to reach the end of this mode.

Discovery mode: This mode allows the user to freely navigate the 3D space by controlling both the direction (left, right, up and down) and speed of the submarine (acceleration and brake). With higher freedom of navigation, the user may explore the space, looking for more details of an interested object. The user may also "collect" fish by driving the submarine closely enough to "catch" one. Hitting a mine will trigger the same effects as in the Game mode, but there is no score reward for avoiding the mines and obstacles.

The movie shows the selected mode with the first person view of the 3D space from the submarine, during which the user may always switch to another mode. The user can also stop the selected mode, or watch and play until the time limit has been reached. At the end, the submarine floats back to the surface through a tunnel, after which a graphical interface is shown so that the user may decide whether to start the movie over again.

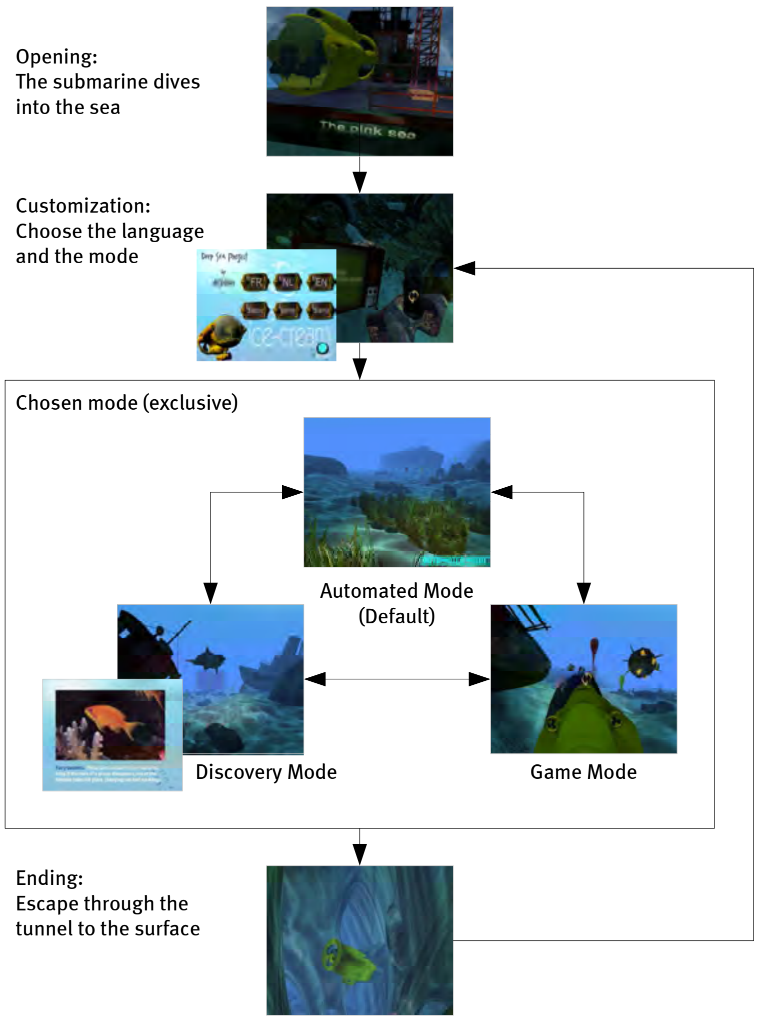


Figure 11.2: DeepSea content structure

11.1.2 Configuration

The prototype utilizes the object oriented features of MPEG-4 to create the 3D virtual space with multimedia objects, such as multiple video and audio streams, 3D graphics, still pictures and text. It also includes lighting effects, graphical interfaces and vibrations as multimedia elements that are distributed over multiple devices. The movie itself is rendered by a Linux PC running a 3D movie engine from de Pinxi which simulates a DVB-MHP platform.

The following devices were used for presentation and interaction:

Primary display: A 42-inch plasma TV (figure 11.3(a) on the next page) is used to present the 3D movie. Two HiFi speakers are connected to and placed near the TV. They are driven by a sound sampler in order to carry realistic and compelling stereo sounds.

Secondary display: An iPronto (figure 11.3(b) on the facing page) is used as a secondary display. It is a small portable computer for controlling home appliances developed by Philips, with a 6-inch touch screen and wireless network connection (Philips, 2003). A UIML based GUI interface was constructed on this display for content navigation and customization, and to "collect" and "review" multimedia objects.

Lights: Four lights were used to present the lighting effects. An orange and a red light are attached behind the primary screen projecting towards the wall. Two stand lamps (figure 11.3(c) on the next page) are placed in the corners. The lighting effects in the virtual 3D space are connected to these physical lights in the room. For example, when the mines blow off, the lamps take part in the effect of the explosion and glare-up as red, orange and white lights; when the submarine dives deeper into the sea, the light is dimmed; when the submarine floats out of the surface, the light becomes bright again.

GamePad controller: A GamePad (figure 11.3(d) on the facing page) controller is used for the navigation control of the submarine. The user needs both hands for the navigation: the left hand to drive the submarine around by pressing left, right, up and down buttons, and the right hand to control the speed by holding and releasing an acceleration button.

LegoMarine: The user may also use a toy submarine for navigation instead of the GamePad. The toy submarine (named LegoMarine, figure 11.3(e) on the next page) can be perceived as the physical counterpart of the submarine in the virtual space. The user may direct the virtual submarine by tilting the toy and speeding it up by squeezing the toy. When the virtual submarine hits a mine or an obstacle, the toy vibrates to give users a tangible feedback.

Before the implementation of the DeepSea application was started, the ideas and concepts of the overall IPML architecture were not mature enough to be put into production. Due to the time pressure also the nature of a cooperative project, the DeepSea application was not fully based on the IPML architecture proposed in previous chapters. However, during the implementation, these design ideas and concepts were gradually becoming clear. The lower level architectural components, such as actions, actors and channels, were put into practice in implementing DeepSea. Several actors based on the concepts presented in chapter 6 on page 65

(a) Plasma TV

(b) iPronto

(c) Lamps

(d) GamePad

(e) LegoMarine

Figure 11.3: Devices used in the DeepSea prototype

and chapter 8 on page 99 were designed: A robotic actor on LegoMarine , a UIML based GUI actor on iPronto, and a lighting actor based on the Velleman K8000 PC interface board (Velleman, 2002). These actors were connected to the de Pinxi 3D movie engine through the channel services (figure 11.4), and these channels were implemented according to the concept described in chapter 7.

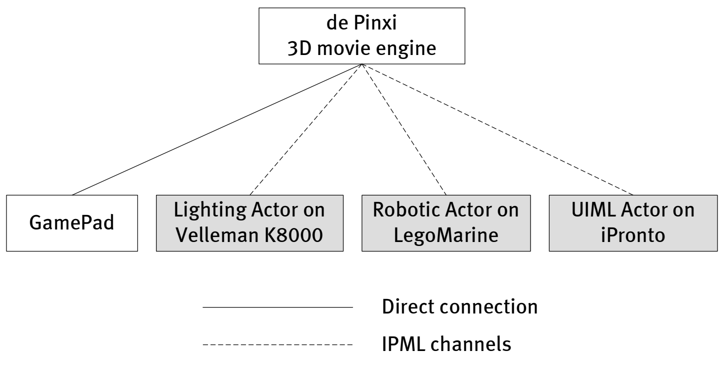


Figure 11.4: IPML contribution to DeepSea

The timing and mapping concepts from the previous chapters were not implemented in DeepSea. Instead, de Pinxi implemented the scheduling mechanisms for the actors in their 3D movie engine by directly changing the source code. The movie engine communicates with these actors by sending action service requests to them and collecting the user interaction events from them. Since the communication protocol is fairly simple, these actors were developed (at Philips Research, Eindhoven) and the movie engine was implemented (at de Pinxi, Brussels) without constant meetings. Before the installation for the user evaluation (see chapter 12 for details), there was only one technical meeting in between to define the communication protocol together. Right before the user evaluation, it was fantastic to see how these distributed actors[1] started talking to the movie engine that was first time delivered to the Philips HomeLab for user evaluation. The concept of loosely coupling the actors through asynchronous channels worked.

Also notice that these software components in DeepSea are programmed in several different languages on different platforms: The 3D movie engine is programmed in C++ and runs on Linux PCs, the lighting actor is programmed in PASCAL, whereas the robotic actor on LegoMarin, the GUI actor on iPronto and the channel services are programmed in Java.

11.2 TOONS in IPML (2003-2004)

After the ICE-CREAM project, the implementation of the IPML design was fully started. Since the content from the TOONS application developed during the NexTV project was available in open standard formats and did not require a proprietary movie engine to playback the video clips, the content was reused to experiment with the IPML design. The content followed the original TOONS scenario (see section 2.1 on page 11 for details), but with few more devices involved to simulate a distributed environment.

The result was a demonstrator that included the following distributed components and devices:

- A movie actor presenting the TOONS movie clips. It is implemented using JMF, and is able to present all audio/video formats that are supported by JMF. The movie actor runs on a Windows PC (pc_1).
- A UIML actor presenting sideshows that are synchronized with the TOONS movie, explaining what is going on in the movie. The UIML actor runs on another Windows PC (pc_2).
- A Linux Infrared Remote Control (LIRC) (Bartelmus, 2003) actor that detects the input events according to a LIRC specification file for a particular remote control, runs on pc_1. A PlayStation 2 remote control was used.
- A robotic actor, Tony 1.5 (figure 11.5(a) on the facing page), and later upgraded to Tony 2.0 (figure 11.5(b) on the next page), standalone.
- A virtual Tony robot, a virtual actor for real Tony, programmed using video clips taken from real Tony, runs on pc_1.
- The IPML director, with timing and mapping engines, runs on pc_2.

[1]They were really distributed, over more than a hundred kilometers from Eindhoven to Brussels.

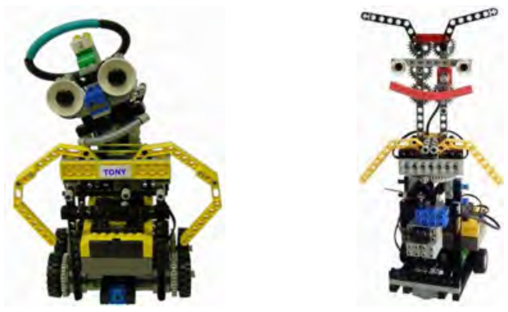

(a) version 1.5 (b) version 2.0

Figure 11.5: Tony upgrated

All software components are programmed in Java. Two Windows PCs are connected over an IP network. The remote control and Tony are connected to pc_1 and pc_2 respectively through infrared sensors that are connected to a serial port on the PC.

This demonstrator has bee set up at the exhibition space of the Department of Industrial Design at TU/e (figure 11.6(a)). It was also exhibited at IST 2004 (Hu, 2004, figure 11.6(b)) and presented at DesForm 2005 (Hu, 2005).

(a) TOONS at ID, TU/e (b) TOONS at IST 2004

Figure 11.6: TOONS demonstrations

11.3 TheInterview (2004)

Taking the TOONS application as a testing bed, the components designed for the IPML system were implemented, including the director with timing and mapping engines. Several hardware and software actors were created. Up to a point, the IPML system was ready for new content and new actors. It was then tried with new content - a movie called *TheInterview*[2] written, designed, directed and shot specially

[2]The movie was made together with Christoph Bartneck.

Scene: *Interview*. Shot: 8
Description: *Murphy drops the papers and looks at Justin "Thank you for ..."*

Scene: *Interview*. Shot: 14
Description: *"yes, the instructions ..."*

Scene:*Interview*. Shot: 8
Description: *"yeah, all that traffic..."*

Scene:*Interview*. Shot: 9
Description: *"Well, thing is..."*

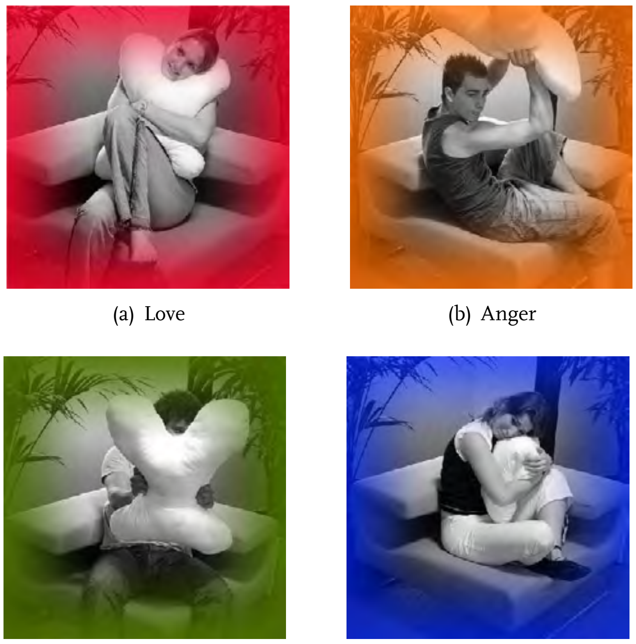
Figure 11.7: TheInterview screen shot descriptions

for this project, in order to conduct a cross cultural study to test the influences of different cultural backgrounds on the user's presence experience in interacting with a distributed interactive movie (see chapter 13 on page 189 for details). Figure 11.7 shows part of the video plan sheet prepared for shooting TheInterview.

The software and hardware actors are reused, but with a different configuration:

- The movie actor on a Windows PC (pc_1), presents the movie to a projected screen.
- the Linux Infrared Remote Control (LIRC) (Bartelmus, 2003) connected to a PlayStation2 remote control through a serial port with an infrared sensor, runs on a Linux PC (pc_2).
- Tony 2.0(figure 11.5(b) on the preceding page), standalone, connected to pc_2 with an infrared sensor connected to another serial port.
- The virtual Tony robot, runs on pc_1, or a wireless display connected to the third computer, depending on the experiment configuration.

The IPML system was put in real use for the first time and tested with more than 50 users, each experiences the movie with a different setting or content three times. The system was robust enough to sustain all these tests with different configurations of users, devices and contents.



(a) Love (b) Anger

(c) Fear (d) Sadness

Figure 11.8: Four selected emotions in Mov'in

11.4 Mov'in - Emotional Pillows (2005)

Since a well designed architecture should not be difficult to be understood, to be extended and to be built new applications upon, the IPML system was handed over to a group of 4 second-year students[3] from the Department of Industrial Design at Eindhoven University of Technology for an 8 week project "Mov'in: Ambient Intelligent Movies" (Menges et al., 2005). The goal was to create movies that not only distribute physical effects to people's environments to enhance their experience, but also sensible to people's movements[4] (hence "Mov'in") and possibly their emotional reactions. The students spent 5 weeks to come up with a concept to design a pillow that would present itself to the IPML system as an actor. The pillow would react to the movie events, showing the emotions in these events by movements and gestures. The users may also react on the emotional outputs from the pillow also by movements and gestures, for example by giving a hug to show love, by slamming it to show anger and by hiding behind to show fear (figure 11.8). The user's emotional input will be picked up by the system to influence what's happening in the movie.

[3]Rutger Menges, Jan v.d. Asdonk, Laurie Scholten and Lilian Admiraal.
[4]This project was also coached by Sietske Klooster, an expert in designing product movements.

During the first 5 weeks four 2-hour workshops were organized for the students to brush up their Java programming skills and to gain some experience with XML based scripting. The architecture of IPML and the APIs were explained to the students by the end of week 5. With basic Java programming skills and some previous experience in programming LEGO mindstorms and microcontrollers, the students were able to present a prototype which proves their concept 3 weeks later. Two actors were used in their final prototype:

- The movie actor that comes with the IPML system, showing a Macromedia Flash movie;
- A robotic pillow that not only shows emotions through movements, but also reacts on the user's emotional movements. Designed and implemented by students themselves.

Figure 11.9(a) shows the "X" shaped emotional pillow and figure 11.9(b) shows what is inside the pillow (one of the arms).

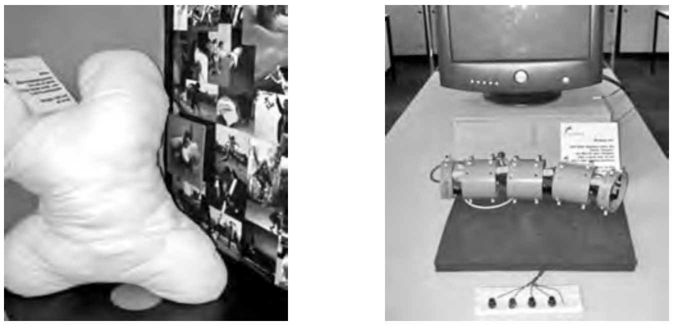

(a) "X" shaped pillow (b) What's inside (one of the arms)

Figure 11.9: Prototype of the emotional pillow actor in Mov'in

11.5 Concluding remarks

The architecture has been mostly implemented in Java. Not counting the earlier implementations of StoryML and testing code, the latest version of the IPML system is a package of 40,542 lines of source code, organized in 324 classes. The architecture has been applied in various projects, from a big EU project together with professional developers, to a small educational project by a team of four second year university students. Each resulted in a working demonstrator or prototype.

The underlying software infrastructure of the implementations varies, including the standard Java Virtual Machine (JVM) from Sun microsystems on both Windows and Linux, the Kaffe JVM for iPAQ with the Familiar Linux distribution, Insignia Jeode JVM for PocketPCs, Tao intent JVM on iPronto, and LeJOS Virtual Machine on Lego Mindstorms RCX. Although they are all implemented in Java, the proposed architecture can also be implemented on a different object oriented platform.

CHAPTER 12

Fun and Presence[1]

Ambient intelligent environments allow multimedia elements to be distributed to multiple networked devices and to bring more compelling entertainment experiences to the end users. This chapter presents the user evaluation results of DeepSea, a distributed interactive movie in such an environment. The movie was presented to single and multiple users with different levels of control and distribution in order to find out the effects on the end user's fun and presence experience. The results suggest that the increased level of control has certain effects on the user's experience, and especially has significant impact on presence. However the effects of distribution are found to be less clear and to be dependent on the involved devices and the content modality. Participation of a second user improves the situation and invokes more fun, but this observation needs to be further verified with more participants. These results are discussed in this chapter in terms of the measurement instruments and the experimental design.

12.1 Introduction

This chapter is going to discuss about the end user's experience, however, the definition of a user's experience is currently still under heavy debate. Karat's definition of an entertainment experience is adapted, as being the experience people are voluntarily going through for pleasure and fun in an entertainment activity (Karat, Pinhanez, Karat, Arora, and Vergo, 2001). Hoonhout (2002) identified seven constructs that are important for and related to the fun experience: enjoyability, attention, curiosity, presence, situational factors and pride. Presence is a more widely used construct and several validated measurement tools are available for it. A detailed discussion of fun and presence is available in section 12.2 on the following page.

The user's fun and presence experience may be influenced by the new interactive technologies. These technologies contain two main aspects: distribution and level

[1]This part of work was done closely together with Hyun-joo Kong. This chapter is based on a paper published in the proceedings of HCI International 2005 (Hu, Janse, and Kong, 2005).

of control. The distributed devices and sensors confront the user with a multitude of media presentations and control possibilities. Instead of watching a movie on a single TV, the user now can experience a story on multiple displays and even the lights in the room react to the events in the story. The effect of such a distributed media presentation on the users is still not completely clear. For example, the recent commercial success of multichannel home audio systems might suggest that multichannel audio improves the user's experience, but it has been shown that simple stereo sound provides just as much presence as six audio channels (Freeman and Lessiter, 2001).

The multimedia standards mentioned above push new interaction technologies forward. The user may interact with the media content using various input methods, such as speech, touch screens and game pads. The classic passive TV watching can be turned into an interactive free exploration, but also intermediate levels of interactivity are possible. All these interactivity levels differ on how much control the user has on the media content. The effects that the different levels of control might have on the user's entertainment experience are not fully understood. Since these technologies become paramount in the domain of networked audiovisual media and home platforms, it appears important to study these effects. However, many previous studies focused on single users whereas TV and movies in real life are often watched together with others. The participation of others might have an important impact on the user's entertainment experience.

Based on the above discussions, three research questions are proposed:

1. What influence does the distribution of media presentation have on the user's fun and presence experience?
2. What influence does the level of control have on the user's fun and presence experience?
3. What influence does the participation of a second person have on the users' fun and presence experience?

Two experiments were conducted based on the DeepSea implementation (see section 11.1 on page 161 for details) to address these questions. The first experiment was designed to answer question one and two, and the second experiment to answer question three. It has to be made clear that the prototype also set the boundaries for the experiment. For example, using a bigger television screen would have influenced some results (Lombart and Ditton, 1997). However, the prototype was designed to represent plausible future home environments.

12.2 Measurements

This study wants to know how the user's entertainment experience is influenced by the features offered in the DeepSea, i.e., the distribution of interactive content elements over devices, different levels of interactivity, and the cooperation of multiple users. Before any sensible measurements are made, it is necessary to understand the constructs of the *entertainment experience* and make them measurable.

12.2.1 Fun and Presence

What is *entertainment* in the first place? Literally, entertainment is an amusement or diversion intended to hold the attention of an audience or its participants. Langer (1977) described entertainment as being “any activity without direct physical aim, anything people attend to simply because it interests them”, while citing Whitehead’s similar definition of entertainment as “what people do with their freedom”. Secondly, what is *experience*? Over the last years, in the field of human-computer interaction and interaction design, many agree that the user experience matters, but with so many definitions of “experience” that it becomes a buzzword. Among these definitions, there are basically three different ways of talking about “experience” (Forlizzi and Ford, 2000):

1. as the constant stream that happens during moments of consciousness,
2. as a story to condense, to remember and to communicate with others, and
3. that has a beginning and an end, and affects the user and the context as a result.

The last one seems more appropriate for our purpose.

These definitions are beyond the scope of this thesis. For the purpose here, entertainment experience can be defined as the experience people are voluntarily going through for pleasure and fun in an entertainment activity (Karat et al., 2001). This gives the study the direction for the experience evaluation, that is, the evaluation of the entertainment experience in the context of DeepSea is narrowed down to a matter of whether the user enjoys the movie and has fun.

Fun is a multidimensional construct. Hoonhout (2002) identified seven different factors that are considered to be important for and related to the fun experience, based on the taxonomy by (Malone and Lepper, 1987), the factors that contribute to pleasure in using consumer products (Jordan, 1998), and the flow experience (Csikszentmihalyi, 2000, 1991). Although these factors are not orthogonal and they contribute to each other, they provide with a closer and more detailed view into the fun experience:

Enjoyability: The degree of enjoyment that users reach when they are voluntarily undergoing an experience that interests them and gives them some amount of pleasure or release.

Attention: Attention is the degree to which a person focuses on the presented media content. It is a cognitive process of selectively concentrating on one thing while deliberately ignoring other things.

Challenge: Levels of challenge are considered to be high in the media contents that stimulate people to think and where the outcome is uncertain.

Curiosity: The tendency of people to seek for something novel. It is a condition for sustained interest and a pre-requisite for people to focus their attention. Sensory curiosity involves, for example, attention-attracting variations and changes in the light, sound or other sensory stimuli of an environment (Malone and Lepper, 1987).

Control: The degree to which the users feel at ease and in control in the environment where the entertainment takes place.

Pride: The degree to which the users feel proud of possessing of the system or consider it for the future use.

Presence: Presence is often referred to as a subjective experience of being in one place or environment, even when one is physically situated in another. It is an illusion of "being there", "it is here" or "being together" (Schuemie, van der Straaten, Krijn, and van der Mast, 2001; Slater, Linakis, Usoh, and Kooper, 1996; Witmer and Singer, 1998). Lombart and Ditton (1997) defined presence in a more general sense as "the perceptual illusion of nonmediation". Freeman, Lessiter, and IJsselsteijn (2001) identified 4 factors that are related to the presence experience:

- *Spatial Presence*, a feeling of being physically located in the virtual space;
- *Engagement*, a sense of involvement with the narrative unfolding within the virtual space;
- *Ecological Validity*, a sense of the *naturalness* of the mediated content;
- *Negative Effects*, a measure of the adverse effects of prolonged exposure to the immersive content.

The presence experience can be influenced by the extent and fidelity of sensory information, match between actions and reactions, the content and the user characteristics (Freeman et al., 2001). The extent and fidelity of sensory information refer to the ability of a technology to produce a sensorial rich mediated environment. Lombart and Ditton (1997) refer to it as the "media form", and these formal characters are those that involve sensory richness and vividness. These variables are for example the number and consistency of the sensory outputs, visual and aural presentation characteristics, and stimuli for other senses.

According to this understanding, the DeepSea application has many features that might contribute to presence. The distribution of the content using multiple displays and lamps can be seen as a media form that could enrich the sensory output; synchronized ambient lighting effects build a bridge between the virtual and the real, and hence may blur the boundary of "being there" and "being here"; the system reactions are not bounded by the user's action space, but extended to the surrounding environment; manipulating the objects in the 3D virtual space introduces increased interactivity and may give the user more feeling of in-control. Distributed presentation and interaction would make it easier for multiple users to cooperate or to compete. Would these features bring more sense of presence as expected, and would the distribution unfortunately break down the user's illusion?

Appropriate measurement instruments are needed to evaluate the contribution of these features to the fun experience, and among other factors, presence in particular.

12.2.2 Measurement instruments

It was decided to only use subjective instruments. Three instruments were used: 1) The Appeal questionnaire from Philips Research (Hoonhout, 2002) covers all the fun factors mentioned above. 2) The Television Commission Sense Of Presence

Inventory (ITC-SOPI) from the UK Independent Television Commission (Freeman et al., 2001; Lessiter, Freeman, Keogh, and Davidoff, 2001) addresses the Presence factor. 3) Structured interviews were conducted after the participants had finished the experiment and questionnaires.

Appeal questionnaire

The Appeal questionnaire was derived from a questionnaire that was being developed in Philips Research for measuring the degree of enjoyability and fun that media content or consumer electronic products provide for the end users. The original questionnaire was in Dutch and it was translated into English for the multinational participants. It contains in total 39 items that cover the following seven factors of the fun experience: Enjoyablity (11 items), Attention (7 items), Challenge (5 items), Curiosity (2 items), Presence (6 items), Control (4 items), and Pride (4 items).

Although the questionnaire was still in the process of validation, it had been used for several projects in Philips Research. Bartneck (2002) used this questionnaire to evaluate the enjoyability of a robotic interface. Stienstra (2003) used this questionnaire to find out how much fun children might have in playing with interactive toys.

In this questionnaire, only few questions are about the Presence factor. Since special attention was going to be paid to presence, the dedicated ITC-SOPI questionnaire was used instead. The items about Presence were removed from the Appeal questionnaire.

Presence questionnaire

ITC-SOPI is one of the few validated questionnaires for measuring the presence experience of both interactive and noninteractive media. From the original ITC-SOPI questionnaire, 15 items were selected, which were considered most applicable (Freeman et al., 2001): Spatial Presence (9 items), Engagement (2 items), Naturalness (1 item), and Negative Effects (3 items).

Although Freeman et al. (2001) suggested that one could not combine the scores for each factor into one overall "media experience", analyzing these factors separately can provide a good insight into the presence experience.

Interviews

Interviews were conducted with prepared questions that address the presentation modes and the role of the presentation device involved in the experiment. The questions are tailored to the experimental conditions. The interview is complementary to the Fun and Presence questionnaires. The participants were also asked for their comments and suggestions, because the feedback would help the developers to improve their design and implementation.

12.3 Experiments and Results

12.3.1 Experiment 1

A 3× (Level of Control) × 3 (Distribution) mixed between/within subjects experiment was conducted. Level of Control was the within participant factor and had the conditions low (LowControl), medium (MediumControl) and high (HighControl). Distribution was the between participant factor and had the conditions None (NoneDistribution), Lighting (LightingDistribution) and Second Display (DisplayDistribution). Due to practical reasons, the Distribution factor was limited to NoneDistribution in the LowControl condition, to LightingDistribution in the MediumControl condition and to DisplayDistribution in the HighControl condition (see table 12.1).

Table 12.1: Conditions of experiment 1

Level of Control	Distribution		
	NoneDistribution	LightingDistribution	DisplayDistribution
LowControl	yes	–	–
MediumControl	yes	yes	–
HighControl	yes	–	yes

Measurements

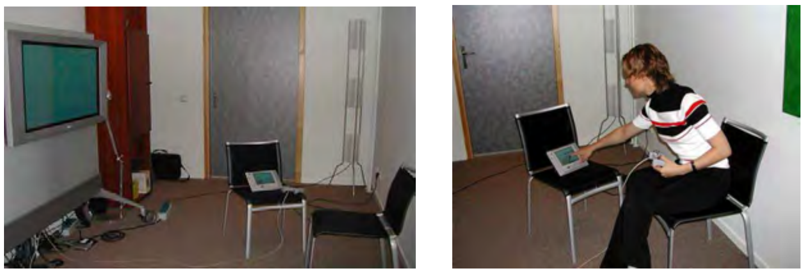
All the items from the Appeal questionnaire were used to measure the fun factors except the Presence items: Enjoyablity (11 items), Attention (7 items), Challenge (5 items), Curiosity (2 items), Control (4 items), and Pride (4 items). The adapted version of the ITC-SOPI questionnaire was used to capture the effects on Presence which had in total 15 items: Spatial Presence (9 items), Engagement (2 items), Naturalness (1 item), and Negative Effects (3 items).

Participants

Eighteen students (9 males, 9 females) from various backgrounds participated in this experiment, nine Dutch, four Ukrainians, two Belorussians, and one Chinese, French and Russian each. 50% of the students had a background in behavioral sciences and the rest in engineering or natural sciences. The average age was 26 years, ranging from 23 to 30 years. The participants watched one to two hours TV per day. Most of them had experience with playing video games and about half of them frequently played games (once or twice a month). They were rather unfamiliar with 3D movies or virtual reality applications.

Setup

The presentation modes in the DeepSea allowed different levels of control. The Automated mode was used for the LowControl condition, the Game mode for the



(a) DeepSea HomeLab setup (b) A user interacting with the movie in DisplayDistribution/HighControl conditions

Figure 12.1: DeepSea Evaluation

MediumControl condition, and the Discovery mode for the HighControl condition. The possibility of switching between different modes during a presentation mode was disabled. In order to catch the effects of distributed lighting, the secondary display was removed from the environment in the MediumControl/LightingDistribution condition. In order to catch the effects of multiple displays, the lighting effects are disabled in the HighControl/DisplayDistribution condition. In all NoneDistribution conditions, only the GamePad was used by the user for interacting with the movie, the secondary display was removed and the lighting effects were disabled. In case the secondary display was needed for starting or customizing the presentation, the operation was done by the experimenters. The toy submarine was not used in this experiment.

The experiment was conducted in a $4m \times 6m$ room in the HomeLab at Philips Research Eindhoven. The 42 inches plasma TV was placed on the wall and 1.5 meters high from the floor. The TV was 3 meters away from the user. When a secondary display was needed, it was placed in front of the user within reach. Except the four lights controlled by the system, there were no any other light sources used, including the natural light. Figure 12.1(a) shows the HomeLab setup.

Procedure

The experiment started with a welcome session and a training movie. The training movie had a similar content, lasted for 3 minutes and allowed the participants to practice all the interaction devices that could be used. Afterwards, they had the opportunity to ask questions about the process of the experiment. Next, the real experiment started, which consisted of the movie with three configurations and two questionnaires (Fun and ITC-SOPI) after each configuration. The order of the conditions was randomized and counterbalanced. At last, a structured interview was conducted with prepared questions. The whole experiment lasted for about one hour. Figure 12.1(b) shows a user interacting with the movie in DisplayDistribution/HighControl condition.

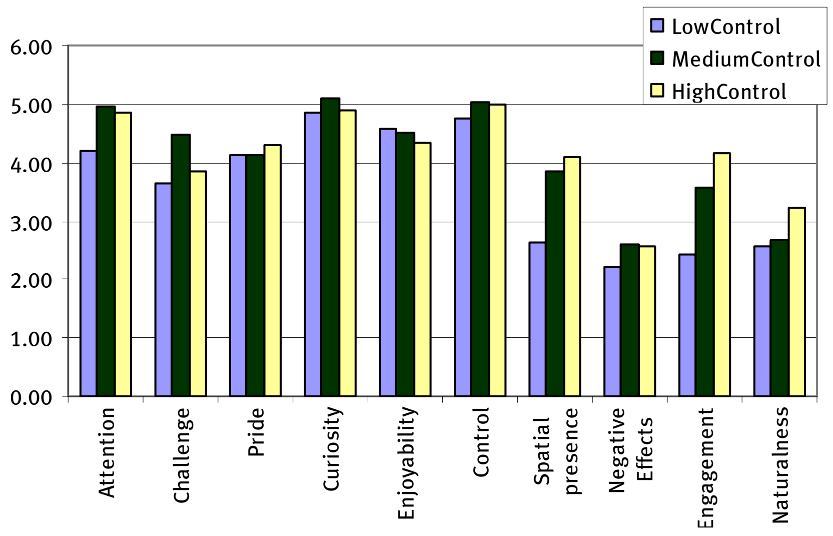


Figure 12.2: Mean scores for all measurements in all Control Conditions

Results

The mean scores for all measurements, including their standard deviations are presented in Table 12.2 on the next page. Due to the fragmented nature of the Distribution factor it was not possible to conduct a single analysis of variance (ANOVA) across all conditions. Instead, one ANOVA for the Level of Control factor and two ANOVAs for the Distribution factor were conducted.

1. Level of Control Effect

Figure 12.2 shows the mean scores for all Level of Control conditions with NoneDistribution.

A 3× (Level of Control) repeated measure ANOVA was conducted. Attention ($F(2,16) = 8.54, p = .01$), Challenge ($F(2,16) = 8.79, p = .01$), Spatial Presence ($F(2,16) = 34.79, p < .001$), Engagement ($F(2,16) = 22.2, p < .001$) and Naturalness ($F(2,16) = 4.96, p = 0.03$) were significantly influenced by the Level of Control.

The Attention scores were significantly lower ($t(8) = 2.64, p = .03$) in the LowControl condition (4.19) than in the HighControl condition (4.84). The Challenge scores were significantly higher ($t(8) = 2.94, p = 0.02$) in the MediumControl condition (4.49) than in the HighControl condition (3.87). The Spatial Presence scores were significantly lower ($t(8) = -6.27, p < .001$) in the LowControl condition (2.64) than in the MediumControl condition (3.85). The Engagement scores were significantly lower ($t(8) = -4.5, p < .001$) in the LowControl condition (2.44) than in the MediumControl condition (3.56). Finally, the Naturalness scores were significantly higher ($t(8) = -4, p < .001$) in the HighControl condition (3.22) than in the MediumControl condition (2.67).

Table 12.2: Mean scores and standard deviations for all measurements

		NoneDistribution		LightingDistribution		DisplayDistribution	
		Mean	std.dev	Mean	std.dev	Mean	std.dev
LowControl	Attention	4.10	0.46				
	Challenge	3.71	0.49				
	Pride	4.08	0.44				
	Curiosity	4.94	0.34				
	Enjoyability	4.62	0.31				
	Control	4.97	0.54				
	Spatial Presence	2.64	0.43				
	Negative Effects	2.22	0.50				
	Engagement	2.44	0.68				
	Naturalness	2.56	0.53				
MediumControl	Attention	4.96	0.38	4.67	0.55		
	Challenge	4.49	0.55	4.64	0.60		
	Pride	4.13	0.25	4.65	0.60		
	Curiosity	5.11	0.37	5.35	0.47		
	Enjoyability	4.53	0.67	5.03	0.60		
	Control	5.03	0.51	4.75	0.53		
	Spatial Presence	3.85	0.31	4.16	0.26		
	Negative Effects	2.59	0.46	2.26	0.43		
	Engagement	3.56	0.73	3.67	0.66		
	Naturalness	2.67	0.71	3.56	0.73		
HighControl	Attention	4.84	0.68			4.63	0.42
	Challenge	3.87	0.66			4.22	0.58
	Pride	4.31	0.41			4.15	0.41
	Curiosity	4.89	0.55			5.40	0.57
	Enjoyability	4.32	0.64			4.52	0.37
	Control	5.00	0.75			4.70	0.45
	Spatial Presence	4.10	0.53			4.23	0.35
	Negative Effects	2.56	0.53			2.44	0.29
	Engagement	4.17	0.43			4.06	0.63
	Naturalness	3.22	0.44			4.11	0.60

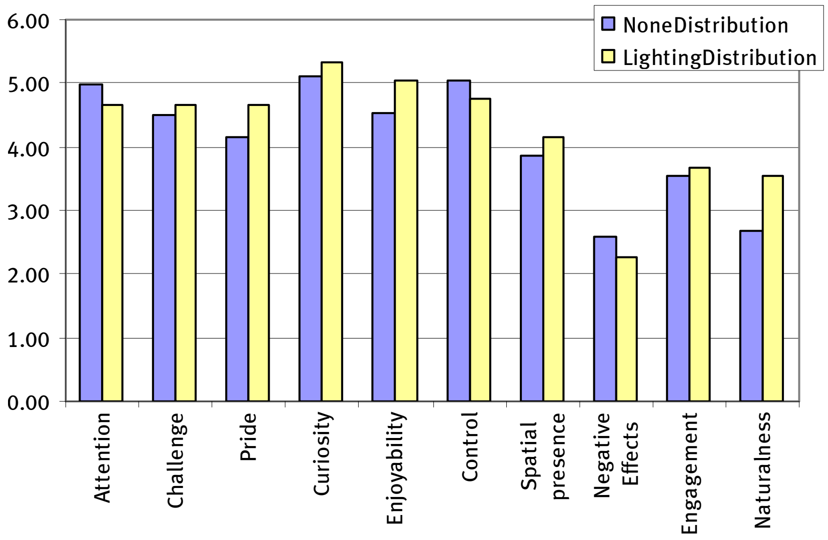


Figure 12.3: Mean scores in the NoneDistribution and LightingDistribution conditions at the level of MediumControl

2. Distribution Effect

Both groups of participants were in the LowControl condition which allows to verify that there were no significant differences between the groups. Therefore, at the same level of MediumControl, NoneDistribution condition was be used for analyzing the effect of LightingDistribution; and at the same level of HighControl, NoneDistribution was used for analyzing the effect of DisplayDistribution (see table 12.1 on page 176).

Lighting distribution: Figure 12.3 shows the mean scores for the NoneDistribution condition and the LightingDistribution condition at the same level of MediumControl.

A 2× (Distribution) ANOVA was performed. Distribution had a significant effect on Pride ($F(1,16) = 5.92, p = .03$), Spatial Presence($F(1,16) = 5.13, p = .04$) and Naturalness ($F(1,16) = 6.92, p = .02$). The Pride scores were significantly lower ($t(16) = -2.43, p = .03$) in the NoneDistribution condition (4.14) than in the LigthingDistribution condition (4.67). The Spatial Presence scores were significantly lower ($t(16) = -2.27, p = .04$) in the NoneDistribution condition (3.85) than in the LightingDistribution condition (4.16). The Naturalness scores were significantly lower ($t(16) = -2.63, p = .02$) in the NoneDistribution condition (2.67) than in the LightingDistribution condition (3.56).

Display distribution: Figure 12.4 on the next page shows the mean scores for the NoneDistribution condition and the DisplayDistribution condition at the same level of HighControl.

A 2× (Distribution) ANOVA was performed. Distribution had a significant effect only on Naturalness ($F(1,16) = 12.8, p < .001$). The scores for Naturalness were significantly lower ($t(16) = -3.58, p < .001$) in the NoneDistribution condition (3.22) than in the DisplayDistribution condition (4.11).

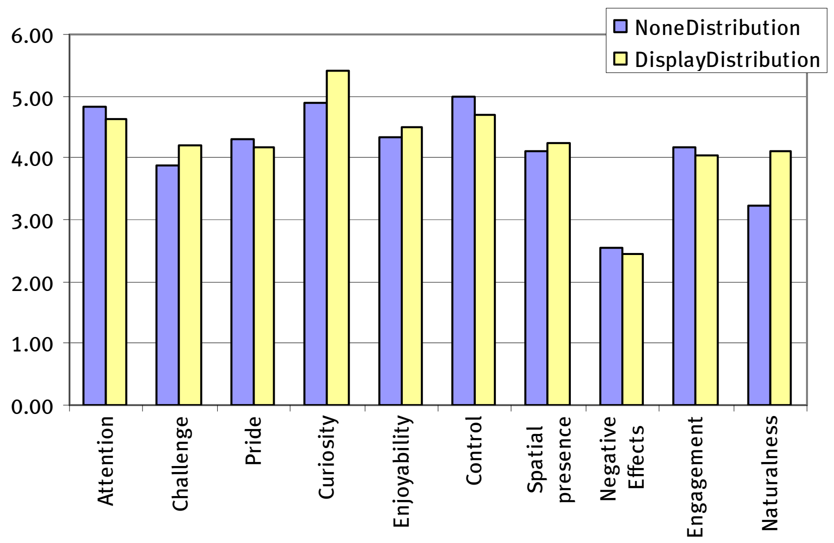


Figure 12.4: Mean scores in the NoneDistribution and DisplayDistribution conditions at the level of HighControl

12.3.2 Experiment 2

A 2× (Number of Users) between participants experiment was conducted. A participant would either interact with the system alone or in collaboration with a second participant.

Measurements

This experiment used the same measurements as used in Experiment 1.

Participants

Twelve participants joined this experiment. They were three couples: 1) 2 males (27 and 26 years old, both have a background in behavioral sciences), 2) 1 male and 1 female (21 and 25 years old, they have a background in computer science and industrial design respectively), 3) father and son (41 and 14 years old, the father works for Philips Research as a software engineer), and six single participants (5 males, 1 female, average age 27 years, most of them have a background in behavior science, engineering or natural science). The participants watched one to two hours TV per day. Most of them had some experience with video games, but the concept of 3D movies and virtual reality applications were new to many of them.

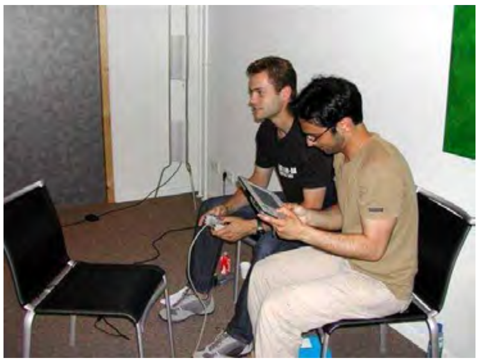


Figure 12.5: Two users cooperating in DisplayDistribution and HighControl condition

Setup

This experiment used the same setup as for Experiment 1 (figure 12.1(a) on page 177), except the fact that both the secondary display and the lighting effects were enabled for the Single and Couple conditions. In the Couple condition, two users were sitting next to each other, 3 meters away from the plasma TV screen. Both GamePad and the Secondary display were placed in front of them within reach. It was up the couples to decide who to use which controlling device. The toy submarine was removed from the system in this experiment.

Procedure

The experiment started with a welcome session and the training movie as in Experiment 1. The real experiment lasted for 20 minutes and the participants were free to start with any of the presentation modes and to switch between different presentation modes. The participants were observed and notes were taken. Videos were recorded and used to substantiate the experimenter notes. Subjects were free to stop the experiment at anytime during the session. After the movie session, the participants were asked to fill out both the Appeal questionnaire and the ITC-SOPI questionnaire. At the end, a structured interview was conducted with the same questions that are also used for Experiment 1. Figure 12.5 shows two participants cooperating in the DisplayDistribution and HighControl condition.

Results

Figure 12.6 shows the mean scores of the two conditions. An ANOVA was performed and all measurements except Challenge and Spatial Presence were influenced by the Number of Users (see table 12.3 on the facing page for the F and p values). Most of the measurements of the fun experience are significantly higher in the Couple condition. It may be concluded that the couples had more fun experience than the singles. As to the effect on the experience of presence, the effect is mixed. The Couple condition has less Negative Effects and Naturalness, but higher Engagement.

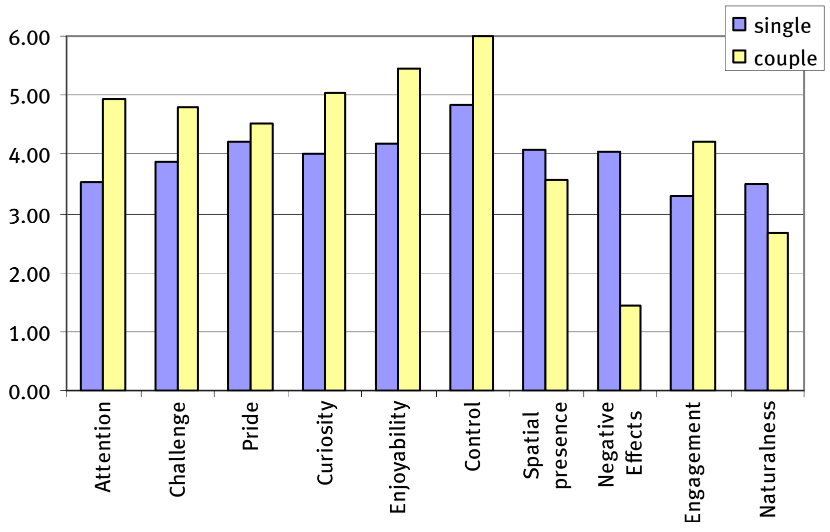


Figure 12.6: Mean scores for all measurements in Single and Couple conditions

Table 12.3: F and p values of ANOVA on the effect of a second participant

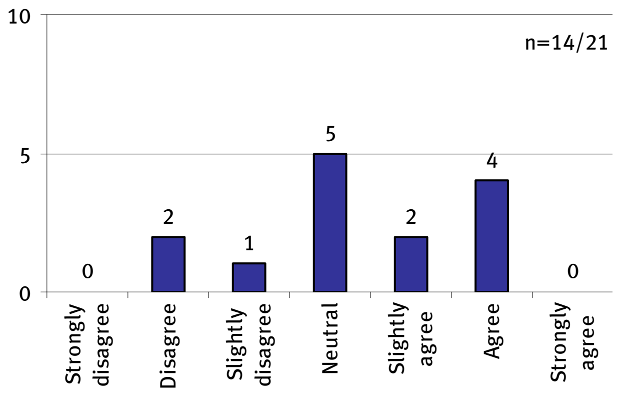
	$F(1,16)$	p
Attention	8.11	0.02
Challenge	0.62	0.45
Pride	10.25	0.01
Curiosity	12.25	0.00
Enjoyability	10.05	0.01
Control	5.88	0.04
Spatial Presence	3.52	0.09
Negative Effects	129.94	0.00
Engagement	9.20	0.01
Naturalness	7.35	0.02

12.3.3 Results from the interviews (Experiment 1 & 2)

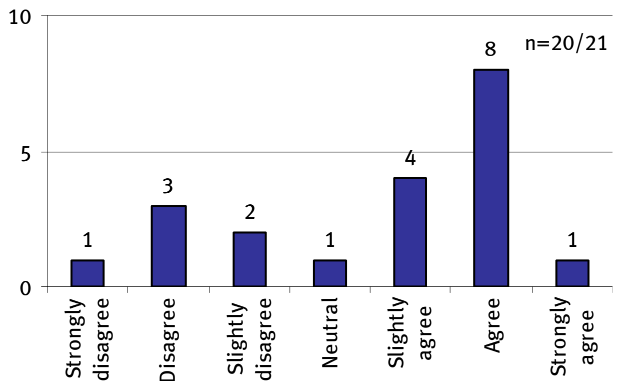
During the interviews the participants were asked for their opinions about every presentation mode and device. One of the questions was to what extent (7 point Likert scale) they agree the statement “I like this mode” about each presentation mode.

Figure 12.7 on the next page shows the distribution of the responses from 21 participants who had experienced the movie in the conditions of LowControl/NoneDistribution, MediumControl/LightingDistribution and HighControl/ DisplayDistribution (9 from Experiment 1 and 12 from experiment 2).

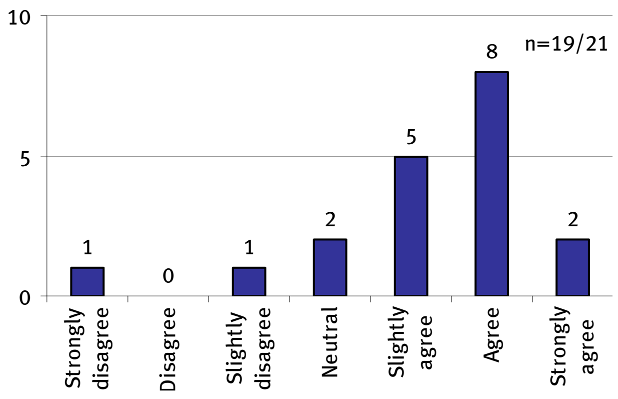
The participants were asked to elucidate further what they liked or disliked. What they liked the most was the freedom to interact with the content in the Discovery mode (HighControl/DisplayDistribution) and many of them considered this mode had the biggest potential among the three modes. But they also complained that it was not easy and comfortable to use two displays at the same time as it kept distracting



(a) Automated mode: LowControl/NoneDistribution



(b) Game mode: MediumControl/LightingDistribution



(c) Discovery mode: HighControl/DisplayDistribution

Figure 12.7: Responses to the question "I like this mode"

the attention. They suggested that there should be more cues for them to switch attention between displays. Another issue was that iPronto was not tightly integrated in the narrative, and participants tended to ignore the interaction with it. For instance when they caught a fish, they just glanced at the iPronto to check the change on iPronto without reading the text about the fish. It could have attracted more attention if the information was presented with audio instead of text.

The strong points of the Game mode (MediumControl/LightingDistribution) mentioned by the participants were the challenge, the excitement, and a simple yet concrete task. They were fascinated by the well synchronized lighting effects. They wanted to have more and varying lighting effects. One of them suggested a blue ambient wavy effect to simulate the underwater environment. Most of the participants said that they would like to have such a lighting system in their homes. With regard to the intensity of the lighting effects, all the female participants wanted to keep it as it was, or decrease it, while all male participants wanted to increase it.

The participants were not very much impressed by the Automated mode (LowControl/NoneDistribution). A few of them liked it for its calm and relaxing presentation. Lacking interactivity and a strong narrative, this mode disappointed many participants. The participants became much more attentive and sensitive to the visual quality of the content. They started noticing the small details of the graphics.

12.4 Discussion

12.4.1 Level of Control effect

The effects of Level of Control on the user's presence experience were expected and the significant differences in Spatial Presence, Engagement and Naturalness confirmed the expectations (figure 12.2). This also confirms the results from many other studies (Agah and Tanie, 1999; Regenbrecht and Schubert, 2002; Waterworth, Waterworth, and R., 2001; Welch, Blackmon, Liu, Mellers, and Stark, 1996). However, Level of Control effects on the measurements in the Appeal questionnaire were less clear. The increase of Attention and Challenge correspond to the findings in the ITC-SOPI presence questionnaire. The more control the users had, the more difficult the task became.

But this did not necessarily result in more fun. No significant differences were found for the other Fun concepts such as Pride, Curiosity, Enjoyability and Control. There are two possibly explanations: either there were indeed not many significant effects caused by increased level of control, or these effects were not caught by the Appeal questionnaire. From the results, it seems that the validated ITC-SOPI is more sensitive than the Appeal questionnaire when measuring the effects of the interactivity. This might be because the ITC-SOPI was designed with the consideration of interactive media, while the Appeal questionnaire was originally designed for passive media and traditional consumer electronic produces. Although it was being tweaked for the interactive media and products, it is still under development and validation. The reliability of the results from this questionnaire needs to be verified and possibly to be improved.

12.4.2 Effects of distribution

Distribution is not a new idea for enhancing the entertainment experience. A good example is the multi channel audio. Over the years, mono sound in early days has been improved to multi channel sound in different levels: stereo (2.0), stereo with bass (2.1), five channel surround (5.0), 5.1 Digital Theater Systems (DTS) surround (five channels with bass). Recently 6.1 DTS surround (six channels with bass) is appearing in the market. Distribution of the sound makes the experience more compelling and more natural because of the nature of the human multidirectional auditory system. Philips recently start producing televisions with Ambilight technology that projects background light from the rear of the television onto the wall, creating a halo around the television, which softly lights the room. Will the distribution of visual content elements enhance the entertainment experience? Both the lighting effects and the secondary display were used for distributing the visual elements.

Less was found than it was hoped for in the results of the questionnaire. Only Naturalness has been affected in both the LightingDistribution condition and the DisplayDistribution condition. This corresponds to previous studies. Lessiter et al. (2001) already suggested that: "The number, extent, and consistency of sensory stimulation (media form variables) are therefore likely to enhance perceived naturalness." However, the other measurements remained suspiciously unaffected in either direction.

From the interviews, it appeared that the distribution of the visual content could distract attention and the participants felt less at ease and in control. This might be a natural result of any kind of distribution - People have to switch their attention from one device to another especially when the stimuli occupy the same sensory channel. This reminds us to keep in mind that the visual channel of the human sensory is not multidirectional in the same way as the auditory sense. Distribution in vision can be distractive. In design, one should be careful with switching visual attention between action spaces. In the LightingDistribution condition, although the lighting effects also occupied part of the users visual sensory, it did not require attention. This explains why Pride and Spatial Presence were significantly higher in this condition and not in DisplayDistribution. In line with this finding, Philips Ambilight televisions are going in a right direction.

The term "displayed environment" appeared in both the Appeal questionnaire and the ITC-SOPI. The participants found it to be difficult to understand especially in the distributed configurations. From the designer's point of view, both the lighting effects and the content elements presented on the secondary display were a part of the movie, hence a part of the "displayed environment". It was also stated clearly in the introductions of both the questionnaires that "We use the term 'displayed environment' here, and throughout this questionnaire, to refer to the film, video, graphics, the virtual world and the physical effects that you have just encountered". This statement was from the original ITC-SOPI questionnaire and was revised to adapt to the distributed configurations. It was possibly not stressed enough in the experiments. In the interviews, it was found that some participants only took the 3D virtual world presented by the plasma display as the "displayed environment" and did not consider all the distributed content as a whole. This might have increased the deviations and hence hampered the reliability of the results.

12.4.3 Number of Users effect

The number of users had a strong effect on almost all measurements. The participation of a second user increased all the measurements of the Appeal questionnaire, but the effects on the Presence measurements were mixed. A possible reason for this could be that a second user takes away attention from the media presentation system and thereby decreases its immersiveness. Both, Spatial Presence and Negative were reduced, even though the reduction of Spatial Presence was not significant, possibly because of the limited number of participants. Furthermore, the two participants had to share certain devices and thereby reduced the number of available stimuli. This has previously been shown to reduce Naturalness (Lessiter et al., 2001).

On the other hand, the second user increased Engagement, which could be explained by the possible presence of a social facilitation effect. Humans tend to try harder in the presence of others. The experimenter's observations during the experiment in combination with the results of the structured interviews lead us to believe that people started with a division of tasks and that they were much more motivated to explore the content in depth.

12.4.4 Content

To observe the effects of the interactivity and the distribution, the content itself was tried to be kept as neutral as possible - to keep the audience not too excited nor too bored by the content itself. The DeepSea application was derived from existing content elements which de Pinxi (2003) originally designed for theme parks and a larger audience. The narrative was reduced to a minimal level, that is, there was not really a story plot in the content. From the interviews, this had resulted in low motivation of the participants to interact with the content, and even boredom.

If the content was designed for production, and not for the experimental sake, this should not have been done. The design of content should also be dedicated specifically to the home environment and not be derived from the content for large audience spectacles and theme parks. The motivation of the participants in both settings is quite different. People carefully plan to go to a multimedia spectacle or to a theater, they usually don't go alone, they don't leave halfway the show unless it is terribly bad and it is a full evening/afternoon commitment. In contrast, behavior at home is much more spontaneous. People don't sit through a full program if they don't like it, they zap, they turn the system off, they make phone calls, they go to the fridge, etc. It takes much more to keep them motivated to stay with the program and to remain enticed to interact with it. The feedback from the interviews showed that it was very important to provide the users with stimuli that motivated them to explore and interact with the content. A clear goal and more opportunities for interaction with objects in the content support these motivational aspects. Support for this finding can also be found in (Malone and Lepper, 1987).

12.4.5 Experimental Design

The distributed lighting effects and graphical elements on the secondary display were considered as a part of the content, but they were however not present in the basic LowControl/NoneDistribution conditions. This might result in arguments whether the content in different configurations are the same. If not, another variable actually had been introduced in the experiment, that is, besides the interactively and the distribution, the content should also have been considered as a variable, although it was tried to keep the content difference between the interaction modes a minimum.

12.5 Concluding remarks

There were high expectations for distributing interactive media in an ambient intelligent environment to bring more compelling entertainment experience to end users. The ambitions of measuring the user's entertainment experience were limited to an operable level, that is, to look at the factors of the Fun experience, and the Presence factors in particular.

These expectations were partially proved to be true in the experimental settings. The increased level of control did have some positive effects on the user's experience, especially had significant impact on the Presence factors. As to the distribution of content elements, the expectations had been lowered by the results from the experiments. The effects were mixed. It also seems to be dependent on the devices to which the content is distributed, the modalities of these distributed content. In this study, distributed lighting seemed to have more positive effects on the user's Fun and Presence experience than distributed displays. Multiple users, or more precisely, two users cooperating improved the situation and invoked more positive experience, but this observation needs to be further verified with more participants.

The evaluation of the DeepSea application provided a lot of valuable feedback, for the next iteration of the design process. The concept was perceived as very promising and interesting by the participants and many said they “liked” it. The evaluation also revealed that there is still a long way to go with regard to the development of methodologies for evaluation, the design of the measurement instruments, the exploitation of the possibilities for interaction with content by users in the home environment, and the exploration of distributing ambiance effects in synchronization with, or as a part of the content.

CHAPTER 13

Culture Matters[1]

Through the evaluation of the DeepSea, certain effects of the level of control and the distribution on the user's fun and presence experience were found. However the directions of the effects were mixed. Especially the effects on presence were less clear than they were expected. This project started questioning what else could caused this. Since the participants of the experiments were from various cultural backgrounds, it was suspected that the user's cultural background could have certain effects on their experiences, and the variance in cultural background could have interfered with other factors.

Based on the full implementation of the IPML architecture, a distributed interactive movie (TheInterview) was developed and a cross cultural study was conducted to test the influences of different cultural backgrounds on the user's presence experience in interacting with this movie.

13.1 Introduction

The user's character (the combination of qualities or features that distinguishes one person from another) is believed to influence the user's feeling of presence. The user's cultural background is often mentioned as such a characteristic (Freeman et al., 2001; IJsselsteijn, Ridder, Freeman, and Avons, 2000). A few cross-cultural presence studies are available (Chang, Wang, and Lim, 2002), but none investigated the relationship between the user's cultural background and presence directly. As Sas and O'Hare (2003, p.527) point out, a "large amount of work has been carried out in the area of technological factors affecting presence", but "Comparatively, the amount of studies trying to delineate the associated human factors determinant on presence is significantly less." This influence of culture on presence is, at this point in time, more of a conjecture than a proven fact, and therefore an empirical study was conducted to investigate the relationship.

[1]This work was done together with Christoph Bartneck, published in the proceedings of the 8th Annual International Workshop on Presence (Hu and Bartneck, 2005, PRESENCE 2005).

13.1.1 Two cultures studied

In absence of a clear definition of what cultural factors may influence presence, a good approach is to include participants from obviously different cultures. Using Dutch and Chinese participants in our study optimized this cultural diversion. Hofstede (Hofstede, 1993) provides an empirical framework of culture by defining several dimensions of culture, such as power distance, individualism/collectivism, masculinity/femininity, uncertainty avoidance and long/short-term orientation. China and Holland differ substantially in all dimensions except uncertainty avoidance (see table 13.1). Power distance, for example, refers to the extend to which less powerful members expect and accept unequal power distributions within a culture. The Dutch rank very low on this dimension, while the Chinese ranks very high.

Table 13.1: Hofstede's (1993) Culture Dimension Scores for Dutch and Chinese

	Dutch	Chinese
Power Distance	38L	80H
Individualism	80H	20L
Masculinity	14L	50M
Uncertainty Avoidance	53M	60M
Long Term Orientation	44M	118H

H = top third, M = medium third, L = bottom third (among 53 countries and regions for the first four dimensions; among 23 countries for the fifth.)

There is another motivation to include Chinese participants in this study. Sacau et al. (2005) reports that agreeableness[2], one of the Big Five personality traits[3] (McCrae and John, 1992), is positively associated with Spatial Presence. The discussion about the connection between personality traits and culture is controversial. Still, the Chinese culture does include a very important concept of "Harmony" (Hé 和) which appears to be closely related to agreeableness. Triandis (2002) believes that "agreeableness may be particularly important in cultures that emphasize interpersonal harmony"; Cheung et al. (2001) even consider harmony as one of the Chinese personality traits and their study shows that it significantly predicts agreeableness. Harmony is so deeply embedded in the Chinese culture that it would be unlikely if it would not result in more agreeableness. Chinese Taoism stresses "harmony with nature" (Tiān Rén Hé Yī 天人和一) and Chinese Buddhism emphasizes "Harmony in six aspects" (Lìu Hé 六和):

- harmony in understanding reaches agreement (Jìan Hé Tóng Jǐe 见和同解),
- harmony in habits brings mutual improvement(Jìe Hé Tóng Zūn 戒和同尊),

[2]Agreeableness is a tendency to be compassionate and cooperative rather than suspicious and antagonistic towards others.
[3]Neuroticism, Extroversion, Agreeableness, Conscientiousness, Openness to experience.

- harmony in physical conditions brings co-habitation (Shēn Hé Tóng Zhù 身和同住),
- harmony in words avoids brawl (Yǔ Hé Wú Zhēng) 语和无诤),
- harmony in interests brings peace (Yì Hé Tóng Yùe 意和同悦) and
- harmony in benefits brings co-existence (Lì Hé Tóng Jūn 利和同均).

Even the word "monk" in Chinese (Héshang, 和尚) itself means literally "Harmony Preacher".

From an application point of view, China currently has one of the most promising and opportune economies. Its vast population and large physical size alone make it a powerful global player. China's gross domestic product (GDP) growth of over seven percent indicates its steaming economic situation. Most Chinese already have access to a television set and the local TV manufacturers satisfy the domestic market. However, technology utilizing presence has not yet been produced or consumed. Awareness of cultural differences in presence may help companies to create better products for different markets.

13.1.2 Distributed interactive media

At the same time, distributed interactive media and their influences on the user's feeling of presence were interesting. A new media era is here: passive television programs become interactive with the red button on your remote control (Bennett, 2004). Video games come with many different controlling interfaces such as dancing mats, EyeToy® cameras, driving wheels and boxing Gametraks™ (In2Games, 2005). The D-BOX® Odyssee™ motion simulation system even introduces realistic motion experiences, which were originally designed for theme parks, into our living rooms (D-BOX, 2005). In the vision of Ambient Intelligence (Aarts and Marzano, 2003), the next generation of people's interactive media experience will not unfold only on a computer or television, or in a head set, but in the whole physical environment. The environment involves multiple devices that enable natural interactions and adapt to the users and their needs.

Such a distributed environment might be perceived differently by users from different cultures. Chua, Boland, and Nisbett (2005) conducted a relevant study in which American and Chinese subjects viewed photographs. The Chinese tended to look at the whole picture and rely on contextual information when making decisions and judgments about what they see, whereas the Americans tend to be analytical and pay more attention to the key or focal objects in a scene. Westerners might, for example, concentrate on the woman in the "Mona Lisa" whilst easterners might pay more attention to the rocks and sky behind her. These results might be of relevance to distributed environments. Chinese participants might also take a more holistic view here while the Dutch might focus more on specific objects. It is expected that the focal view on specific objects would find visually distributed presentations to be more distractive than the holistic view. Hence that the Dutch participants would feel less engaged and less natural than the Chinese participants would.

13.1.3 Embodiment and Interaction

The distribution of interactive content to multiple devices might also have a negative effect. It might increase the complexity of interaction. The environment together with the interactive content may become difficult to understand or control. To relieve the situation, embodied characters, such as eMuu (Bartneck, 2002) or Tony (Bartneck and Hu, 2004), may be used to give such an environment a concrete face. These characters have a physical embodiment and may present content through their behavior and interact with the user through speech and body language.

Moreover, the influence of embodiment on the user's presence experience seems unclear. On the one hand, embodiment extends the distributed content from an on-screen virtual environment to a physical environment. The physical embodiment improves the content's liveliness and fidelity by stimulating more senses of the user. This might result in an increased feeling of presence (Lombart and Ditton, 1997). On the other hand, the physical embodiment may transfer more attention from the virtual environment to the physical environment. The physical embodiment may remind the user of its existence in this world and may breakdown the illusion of *being there* and hence could result in less feeling of presence (Freeman et al., 2001). The division of attention in itself might also have such an effect.

To control interactive content, the user requires interaction devices. A physical embodiment would invite direct manipulation. A robot could, for example, ask the user to touch its shoulder in order to select an option. Interaction with a virtual on-screen character may favor the use of a remote control. Embodiment in interactive media can therefore not be studied without considering the interaction method. Therefore two interaction methods were included in the study.

13.1.4 Research questions

In this framework of interactive distributed media the following three research questions were proposed:

1. What influence does the user's cultural background have on the users' presence experience when interacting with distributed media?
2. What influence does the embodiment of virtual characters have on the users' presence experience?
3. Would direct touching of the presented content objects bring more presence than pressing buttons on remote controls?

13.2 Experiment

A 2 (Culture) × 2 (Embodiment) × 2 (Interaction) mixed between/within subjects experiment was conducted(see figure 13.1 on the next page). Interaction and culture were the between participant factors. Interaction had the conditions RemoteControl and DirectTouch, and culture had the conditions Dutch and Chinese. Embodiment was a within participant factor. Embodiment had the conditions ScreenAgent and Robot.

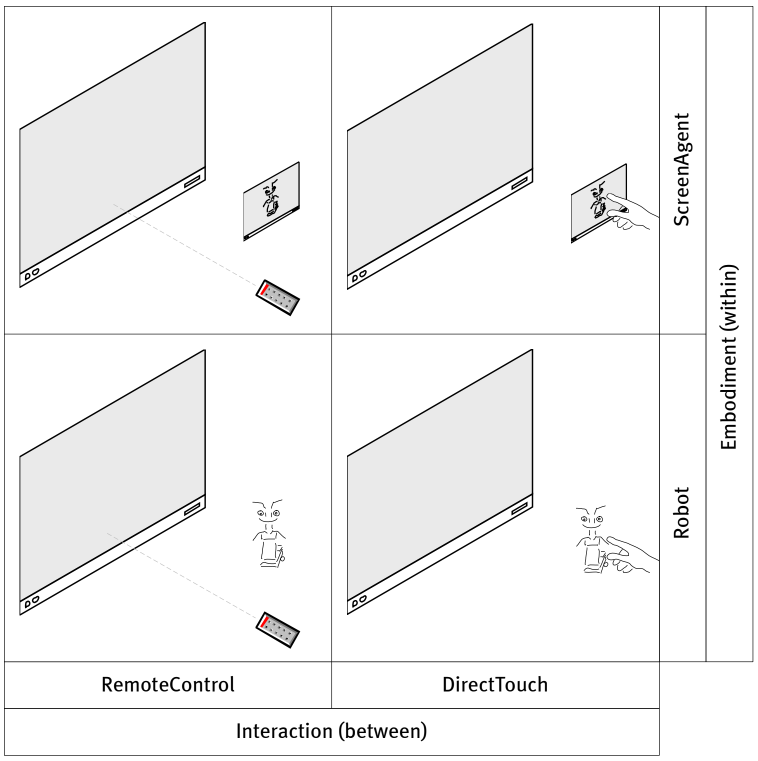


Figure 13.1: Conditions of the experiment with Chinese and Dutch participants

13.2.1 Measurements

The original ITC-SOPI (Lessiter et al., 2001) questionnaire was again used. But different from the ITC-SOPI questionnaire used in the user evaluation of the DeepSea application, only the definition of the *Displayed Environment* in the introduction was adjusted to include the robot/screen character. The Chinese participants had a good understanding of the English language and therefore no validated translation was necessary. The questions remained unchanged and are clustered into four groups:

1. *Spatial Presence*, a feeling of being located in the virtual space;
2. *Engagement*, a sense of involvement with narrative unfolding in virtual space;
3. Ecological validity, a sense of the *naturalness* of the mediated content;
4. *Negative effects*, a measure of the adverse effects of prolonged exposure to the immersive content.

13.2.2 Participants

Nineteen Chinese and twenty-four Dutch participants between the age of 16 and 48 (14 female, 29 male) participated in the experiment. Most of them were students and teachers from Eindhoven University of Technology, with various backgrounds in

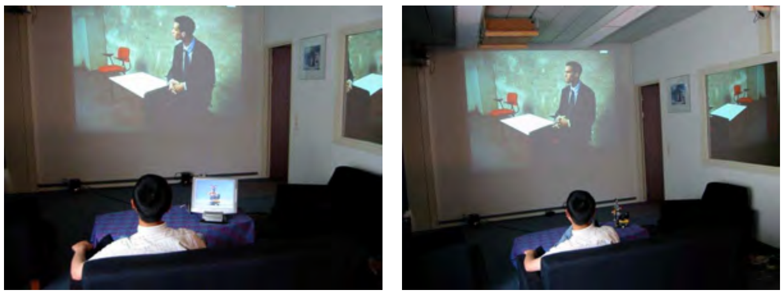

(a) ScreenRemote/ScreenTouch (b) RobotRemote/RobotTouch

Figure 13.2: Experiment setup

computer science, industrial design, electronic engineering, chemistry, mathematics and technology management. The Chinese participants lived no longer than two years in Holland. All participants had a good command of the English language and were frequently exposed to English media, such as movies, web pages, news papers and TV shows.

13.2.3 Setup

Three computers were connected to the public network of Eindhoven University of Technology, with a bandwidth of $10M$ b/s, to serve three presentation terminals. One laptop computer (Windows XP Professional, $1.4G$ HZ Pentium processor, 512 MB DDR memory) was connected with a projector to serve the output of a movie actor to a projected screen. The second computer had the same hardware configuration, but the operating system was Linux 2.6 (Fedora Core 2). Two interaction devices were connected to this computer: the real "Tony" (see figure 11.5(b) on page 167) through an infrared tower connected to the serial port; and the Linux Infrared Remote Control (LIRC) actor detecting the input from a PlayStation 2 remote with an infrared sensor connected to another serial port. It also ran the IPML director that read the movie script and scheduled the presentation tasks for the actors. The third computer (Windows XP, $1G$ HZ Pentium processor, 256 MB DDR memory) was connected to a Philips DesXcape Smart Display through a firewalled local wireless network to serve the virtual Tony actor to a touch screen (also see section 11.2 for details).

The experiment took place in a living room laboratory (see figure 13.2). The participants were seated on a couch in front of a table. The coach was $3.5m$ away from the main screen, which was projected onto a wall in front of the participant. The projection had a size of $2.5m \times 1.88m$ with 1024×768 pixels. The second screen was located $0.5m$ from the coach, standing on the table. The secondary screen was $30cm \times 23cm$ with 1280×1024 pixels LCD touch-screen.

In the ScreenAgent conditions, the secondary screen displayed a full screen agent of the robot. In the Robot conditions, the secondary touch screen was replaced with the Lego robot that had about the same height. The behavior of the screen based agent and the Lego robot were identical. They played the role of a TV companion by looking

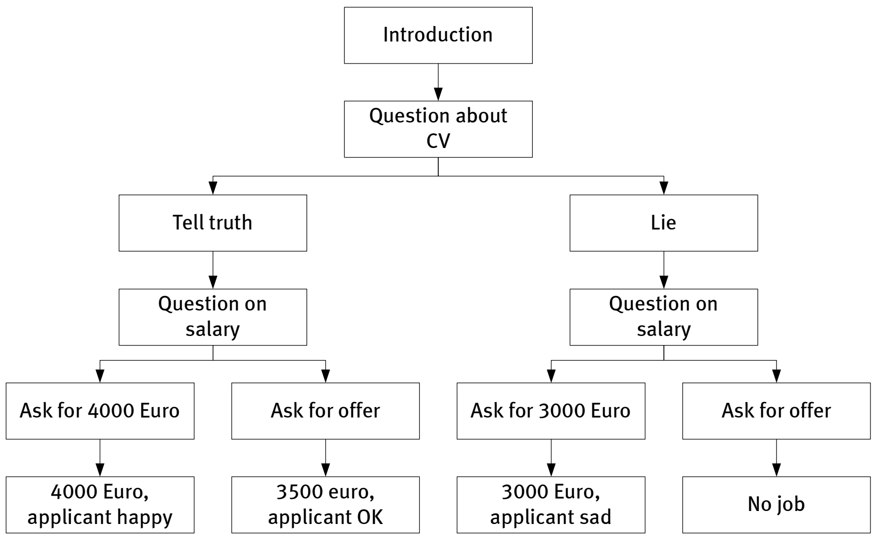


Figure 13.3: Storyline of TheInterview

randomly at the user and the screen, but always looking at the user while speaking. Speakers were hidden under the table to produce synthetic speech. At the start of every movie, the character introduced himself and its role.

Even though a particular media content can be acceptable in one culture, in another can be perceived as inappropriate, rude or offensive (Lu and Walker, 1999). Therefore the movie was designed to be culturally neutral. The movie had an international cast: the applicant and the employer by two Dutch, the secretary by an American, and the passer-by by a Chilean. The actors spoke English. This study does not investigate the influence of media content on presence and therefore the story and movie cuts were neither too exciting nor too boring for both Dutch and Chinese participants. Or they might have masked the effects of culture and embodiment.

The interactive movie, about 10 minutes, was about a job interview. The participants had to make decisions for the applicant. The storyline was discussed with several Chinese and Dutch people to assure that the actions of the characters would be plausible in both cultures. The movie had two decision points, which resulted in four possible movie endings (see figure 13.3). The participants could chose different options during the decision points. At every decision point camera would zoom in on the applicant's forehead (see figure 13.4 on the next page). The actor then cycled through two options in his mind. He looked first to the left and thought aloud about one option, before he looked right and thought aloud about the second option. Then the screen agent or the robot explained the interaction possibilities to the user. In the RemoteControl conditions the screen would show one icon on the left and a different icon on the right. The icons were identical to two icons on the remote control. In the DirectTouch conditions, the participant had to touch the left or the right shoulder of the screen agent or the robot to make the decision.

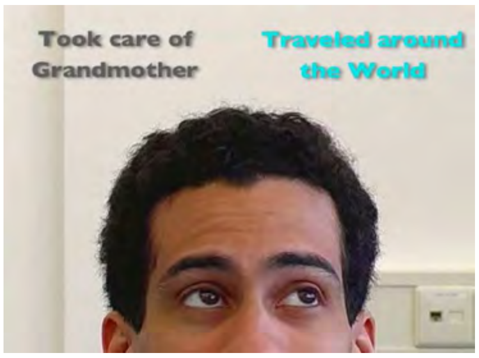


Figure 13.4: A decision point

13.2.4 Procedure

After reading an introduction that explained the structure of the experiment the participants started with a training session. In this session, the participants watched an unrelated interactive movie that only had one decision point, during which the participants could make the decision using a remote control. Afterwards, they had the opportunity to ask questions about the process of the experiment. Next, the participant was randomly assigned to one of the between-participant conditions, that each consisted of two movies in a random order and a questionnaire after each movie. Overall the within and between conditions were counterbalanced. The participant received five Euros for their efforts.

13.3 Results

The mean scores for all measurements, including their standard deviations are presented in table 13.2 on the facing page and graphically in figure 13.5 on page 198.

A 2(Culture) $\times$ 2(Embodiment) $\times$ 2(Interaction) repeated measures ANOVA was conducted. Interaction had no significant influence on any of the measurements. Embodiment and culture both had significant influence on almost all measurements (see table 13.3 on page 199).

Interaction was removed as a factor from the further analyses since it had no effect on the measurements. The means for all remaining conditions are summarized in figure 13.6 on page 198 and were used as the basis for further analysis.

Paired Sample t-Tests were performed across both culture conditions. The measurements for Spatial Presence were significantly ($t(42) = 2.235$, $p = 0.031$) higher in the ScreenAgent condition than in Robot. Negative Effects were significantly ($t(42) = 2.38$, $p = 0.022$) higher in the Robot condition than in ScreenAgent.

Independent Samples t-Tests were performed. All measurements between the Dutch and the Chinese participants differed significantly, except that Engagement in the screen conditions just missed the significance level ($t(41) = 2.007$, $p = 0.051$).

Table 13.2: Mean and standard deviation for all measurements

Embodiment	Culture	Interaction	Measurement	Mean	Std.Dev.
ScreenAgent	Chinese	RemoteControl	Spatial Presence	3.08	0.18
			Engagement	3.35	0.37
			Naturalness	3.17	0.32
			Negative Effects	1.96	0.55
		DirectTouch	Spatial Presence	2.79	0.37
			Engagement	3.28	0.41
			Naturalness	2.92	0.61
			Negative Effects	1.83	0.52
	Dutch	RemoteControl	Spatial Presence	2.56	0.29
			Engagement	3.17	0.51
			Naturalness	2.75	0.50
			Negative Effects	1.46	0.43
		DirectTouch	Spatial Presence	2.44	0.45
			Engagement	2.84	0.58
			Naturalness	2.58	0.74
			Negative Effects	1.50	0.36
Robot	Chinese	RemoteControl	Spatial Presence	2.99	0.20
			Engagement	3.33	0.24
			Naturalness	2.73	0.17
			Negative Effects	3.28	0.42
		DirectTouch	Spatial Presence	2.72	0.59
			Engagement	3.22	0.43
			Naturalness	3.08	0.32
			Negative Effects	3.35	0.52
	Dutch	RemoteControl	Spatial Presence	2.51	0.25
			Engagement	3.07	0.61
			Naturalness	2.55	0.66
			Negative Effects	2.9	0.40
		DirectTouch	Spatial Presence	2.26	0.41
			Engagement	2.86	0.56
			Naturalness	2.42	0.59
			Negative Effects	2.52	0.82

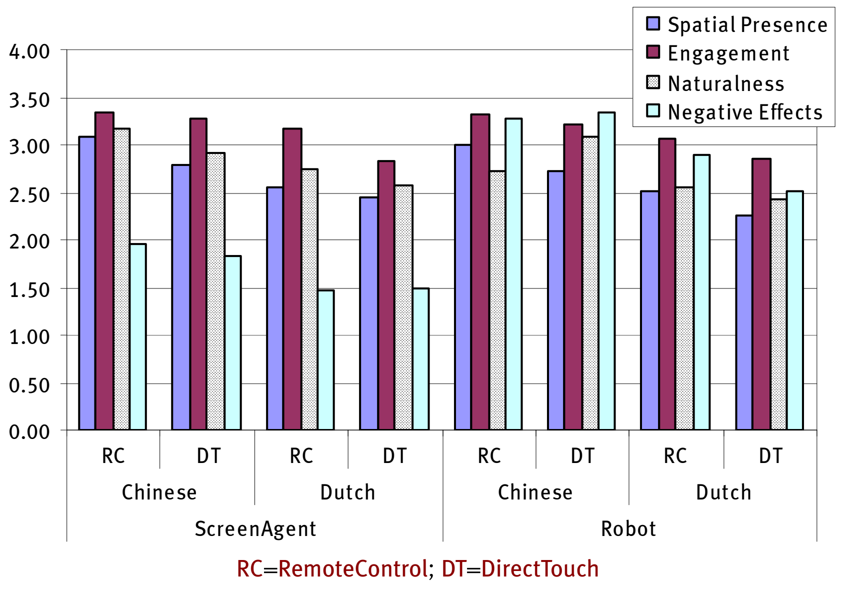


Figure 13.5: Means of all measurements in all conditions

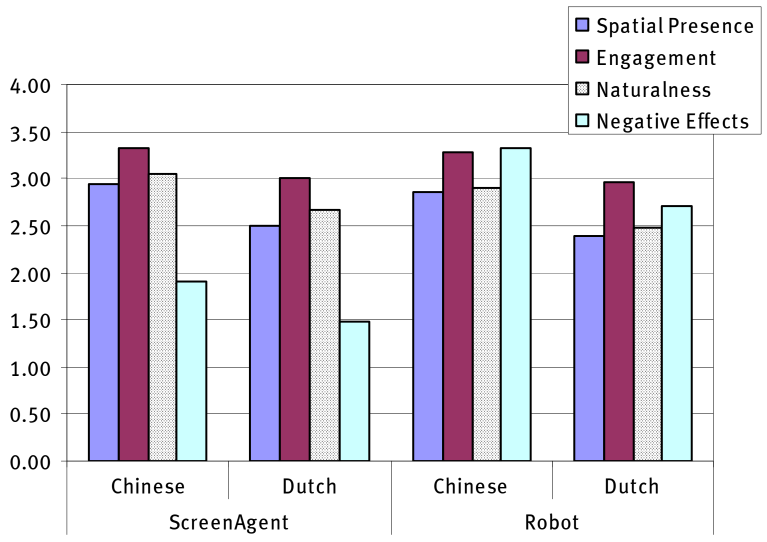


Figure 13.6: Means in the culture and embodiment conditions

Table 13.3: F and p values for culture and embodiment

Factor	Measurement	F (1,39)	p
Embodiment	Spatial Presence	4.789	0.035
	Engagement	0.515	0.477
	Naturalness	4.335	0.044
	Negative Effects	119.973	0.001
Culture	Spatial Presence	19.490	0.001
	Engagement	4.962	0.032
	Naturalness	7.494	0.009
	Negative Effects	24.491	0.001

13.4 Discussion

13.4.1 Culture effects

The participants' cultural background clearly influenced the measurements. Chinese participants perceived more presence than Dutch participants did in all conditions. This result is in line with the results of Chua et al. (2005). It appears that the Chinese take a more holistic view thereby appreciating the distributed media presentation more than the Dutch. Like Chua et al. (2005) predicted, the Chinese and Dutch allocated attentional resources differently as they viewed the distributed environment. East Asians are known to live in a relatively complex social networks (Nisbett and Masuda, 2003). It is essential for them to consider the context in which events happen. Also the results of Sacau et al. (2005) appear to be confirmed. The more agreeable Chinese perceived more spatial presence. One might suspect that the more agreeable attitude of the Chinese might have let them simply to be more polite in answering the questionnaire. The measurements show that they also gave higher scores to Negative Effects and therefore did not simply respond politely.

None of Hofstede's (1983; 1993; 1988) culture dimensions appear relevant to presence at first sight. However, one might speculate that the long-term orientation in Chinese culture would result in more patience and tolerance towards imperfections. In our case the Chinese participants might have more easily tolerated the noise emitted by the robot and the occasional visibility of a microphone in the movie. Further studies are necessary to investigate this issue.

13.4.2 Embodiment effects

The influence of embodiment on all measurements does not conform to the expected results defined in the construct of presence. According to Lessiter et al. (2001), "Whilst in the current study Negative Effects was not strongly correlated (positively or negatively) with Engagement or Ecological Validity, it was significantly but modestly (and positively) related to Sense of Physical Space".

However, in the results Spatial Presence and Naturalness are higher in the ScreenAgent condition, while Negative Effects were higher in the Robot condition. Negative Effects appear to have been affected by something else than presence.

During the experiment, the robot's motor emitted noise, which caused the participants to look at it. A moving physical object is potentially dangerous and hence attracts attention. Clearly, the robot emphasized the participants feeling of being in the room and not in the movie and thereby reducing the presence experience. The screen character did not emit noise and is unable to pose a physical danger to the user. It therefore did not attract as much attention as the robot did. One could have played synthetic noise for the screen robot to equalize the variable of motor-based noise.

The participants frequently switched between looking at the movie and the robot and hence their attention was divided. This switching made it hard for the users to stay focused and might have caused the high negative experience. Eggen, Feijs, Graaf, and Peters (2003a) showed that a divided attention space reduces the users immersion. Further research is necessary to determine if divided attention increases the negative effects of multiple displays. The extra costs necessary to build and maintain a robot for an interactive movie appear unjustified in relation to its benefit.

13.4.3 Effects of direct touching

The interaction methods (using a remote control or touching directly) had no influence on the measurements. The participants did not experience more or less presence when they interacted with a remote control or with the screen/robot directly. This is to some degree surprising, since the participants had to move actively to interact with the screen agent or the robot by leaning forward and touching it directly, where the actions and reactions are tightly coupled. With the remote control, the participants remained leaned back. The necessity to make a choice might have overshadowed the difference in physical movement. To create a compelling sense of presence it might be useful to pay more attention to the physical output than to the input.

13.4.4 Future Research

In this study, several factors were investigated besides the cultural background of the participants. The Chinese participants in this study have been living in the Netherlands and might therefore form a non-representative group of Chinese. They might have been to some degree westernized, but one could speculate that the effects observed in this study might be even be stronger for Chinese participants that also live in China. This experiment would need to be run in China to gain full understanding of this issue. Furthermore, this study investigated several factors besides the cultural background of the participants. It might be beneficial to conduct a dedicated study on the influence of culture on presence. Such a study could then also cover more than the two cultures investigated in this study. In addition, it appears necessary to further connect the results of such a study to existing results in other research areas, such as cross-cultural communication studies. Qualitative interview might help to gain better insights into social and cultural viewpoints of the participants in relation to presence.

CHAPTER 14

Conclusions

This chapter presents the conclusions related to the design process itself. After that the resulting design is reflected on.

14.1 Reflection on the design process

If back to the early phase of the NexTV project, the initial focus was on the user. It was clear that the technology for interactive TV was becoming available and Philips was looking for new applications. That was how this PhD design project started. In this phase, I worked closely with Magdalena Bukowska and other project partners to make the user requirements explicit. The scenarios proposed by the children, being the intended users, revealed two wishes: 1) An active role for the user; 2) multiple input and output devices including robots. The results and viewpoints were adopted by the other partners, notably, FhG FOKUS and NOB. In this phase. software design was considered the main tool to perform the first exploration, bring distributed applications alive.

NexTV ended with interesting demonstrators based on the TOONS movie, including a StoryML implementation. The ICE-CREAM project picked up the technical challenges of distribution as well as evaluation studies with users. DeepSea and the first version of IPML are amongst the results of the ICE-CREAM project. After that the project continued in the Department of Industrial Design of Eindhoven University of Technology towards a full IPML implementation, taking it one step further with other demonstrators including “TheInterview” application.

The following observation and conclusions seem justified.

- The design process has been an iterative one with three main iterations, where each cycle produced in at least one complete demonstrator.
- User involvement occurred at specific phases in every design cycle, see section section 2.1, chapter 12 and chapter 13.

- The technical design work was indispensable for these requirements and the user inputs which were based on real working demonstrators.
- Conversely, the requirements obtained were essential for each next design iteration. For example, the need for more complex narratives rather than only linear structures became clear at the end of NexTV.
- After ICE-CREAM, the DeepSea application showed the need for an open architecture. The intertwining of the content (3D movie) and the dedicated software platform did not allow for clear supplier roles such as content vendors and platform vendors.
- Running the technical system with real users has an added value for the technical work as well: it reveals problems such as software bugs and performance bottlenecks, not easily found when the designer tests the software in his own laboratory.
- To prove that the architecture itself is easy to use for new applications, someone else would have to build an application. This is precisely what happened: Rutger Menges, Jan v.d. Asdonk, Laurie Scholten and Lilian Admiraal, four 2nd year TU/e students created Mov'in where interactive pillows were used to explore "emotional" interaction.
- For the technical design work the ABD method turned out to be useful for identifying and reusing architectural patterns. It provides an architectural view which would not be covered by objects and patterns alone.
- The formal methods used, notably Object-Z, Broy's component based framework, and the Petri net based OCPN and ASE were helpful: They are abstract enough to make mechanisms explicit in a clear way and at the same time they are concrete enough to see the structure of the implementation.
- It was not possible to formalize every aspect of the system. The system is far too complex for that. Yet several of the most important design concepts were covered by formalizing them in a pragmatic way.
- Several formalisms and several types of "syntactic sugar" had to be used to keep things manageable. As an example, an earlier version of part II tried to describe the build-up of connect topologies and the runtime concurrent behavior of these topologies in one specification using Timed Communication Object-Z (TCOZ). It became so complex that the formalization was not helpful anymore.

14.2 Reflection on the resulting design

With respect to the resulting design, all the design decisions have been mentioned already in part II and part III. Here only the main conclusions are given.

- On top of existing network technologies and platform architectures, a generic architecture has been designed to enable playing IPML in a networked environment with user preference and dynamic configurations taken into account.

- The architecture has been implemented and tested in Java. In total 40,542 lines of Java source code created 324 classes. The architecture has been used in experiments with up to five different hardware platforms: desktop PCs, tablet computers, handheld devices (iPAQ, iPronto), LEGO Mindstorms RCX and programmable micro controllers such as Velleman K8000. Although it was all implemented in Java, the underlying software infrastructure varies, including the standard JVM from Sun microsystems on both Windows and Linux, the Kaffe JVM for iPAQ with the Familiar Linux distribution, Insignia Jeode JVM for PocketPCs, Tao intent JVM on iPronto, and LeJOS Virtual Machine on Lego Mindstorms RCX.

- During the design process existing scientific and engineering results were imported and used on every possible occasion. Examples are SMIL and XML as the basis for IPML, CORBA for communication channel services, JINI for service registration and lookup, MPEG-2, MPEG-4, MP3 and QuickTime for content elements, Object-Z, Broy's component based framework and Petri nets for formalization, design patterns and object oriented design for software structures, Hardman's taxonomy of time concepts in multimedia applications, research studies on the experience of presence and fun, and Hofstede's theory of culture dimensions.

- There are also contributions to the body of scientific and engineering results (beyond the design results themselves, of course). Examples are the method of rapid robotic prototyping (section 2.4), the extension of Petri nets as action synchronization engine, and new architectural patterns such as Timed Action (section 6.1), Synchronizable Object (section 6.2), Real-time Channel (section 7.3) and Streaming Channel (section 7.4).

- The results of chapter 12 offer preliminary insights on how distribution, level of control and number of users influence user's fun and presence experience in an AmI movie, notably the influence of the level of control and the number of the users on their experiences. The influence of distribution on the user's experiences is also observed and there can be an effect depending on the type of the distributed content and how the distribution is arranged.

- An effect of cultural background upon the perception of presence in a distributed setting (chapter 13) was found. As far as known this is the first study to experimentally confirm this long-standing conjecture.

Bibliography

E. Aarts. Ambient intelligence: a multimedia perspective. *IEEE Multimedia*, 11(1): 12–19, 2004. ISSN 1070-986X. doi: 10.1109/MMUL.2004.1261101.

E. Aarts and S. Marzano. *The New Everyday: Views on Ambient Intelligence*. Uitgeverij 010 Publishers, 2003.

E. Aarts, R. Harwig, and M. Schuurmans. Ambient intelligence. pages 235–250, 2001.

G. D. Abowd. *Formal aspects of human-computer interaction*. PhD thesis, Oxford, UK, UK, 1991.

A. Agah and K. Tanie. Multimedia human-computer interaction of presence and exploration in a telemuseum. *Presence: Teleoperators and Virtual Environments*, 8(1): 104–111, 1999.

C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

C. Alexander, S. Ishikawa, and Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

H. Alexander. Structuring dialogues using CSP. pages 273–295, 1990.

J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/182.358434.

J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.

J. F. Allen and P. J. Hayes. Moments and points in an interval-based temporal logic. *Computational Intelligence*, 5(4):225–238, 1990. ISSN 0824-7935.

J. Ayars, D. Bulterman, A. Cohen, K. Day, E. Hodge, P. Hoschka, E. Hyche, M. Jourdan, M. Kim, K. Kubota, R. Lanphier, N. Layaïda, T. Michel, D. Newman, J. van Ossenbruggen, L. Rutledge, B. Saccocio, P. Schmitz, W. ten Kate, and T. Michel. Synchronized multimedia integration language (SMIL 2.0) - [second edition]. W3C recommendation, 2005. URL http://www.w3.org/TR/2005/REC-SMIL2-20050107.

F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi. The architecture based design method. Technical Report CMU/SEI-2000-TR-001, The Software Engineering Institute (SEI), Carnegie Mellon University, 2000.

B. Bagnall. *Core LEGO MINDSTORMS Programming: Unleash the Power of the Java Platform*. Number 10/2/2002. Prentice Hall PTR, 2002.

B. Bailey, J. A. Konstan, R. Cooley, and M. Dejong. Nsync – a toolkit for building interactive multimedia presentations. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 257–266, New York, NY, USA, 1998. ACM Press. ISBN 0-201-30990-4. doi: 10.1145/290747.290779.

R. Barrett and S. J. Delany. OpenMVC: a non-proprietry component-based framework for web applications. In *The 13th international World Wide Web conference on Alternate track papers & posters*, pages 464–465, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-912-8. doi: 10.1145/1013367.1013527.

C. Bartelmus. Linux infrared remote control, 2003. URL http://www.lirc.org.

C. Bartneck. *eMuu - An Embodied Emotional Character for the Ambient Intelligent Home*. PhD thesis, Eindhoven University of Technology, 2002.

C. Bartneck and J. Hu. Rapid prototyping for interactive robots. In F. Groen, N. Amato, A. Bonarini, E. Yoshida, and B. Kröse, editors, *The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 136–145, Amsterdam, 2004. IOS press.

L. Bass, R. Pellegrino, S. Reed, S. Sheppard, and M. Szczur. The Arch model: Seeheim revisited. In *CHI 91 User Interface Developpers' Workshop*, Seeheim, Germany, 1991.

L. Bass, R. Faneuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, R. Seacord, S. Sheppard, and M. R. Szczur. A metamodel for the runtime architecture of an interactive system: the uims tool developers workshop. *ACM SIGCHI Bulletin*, 24(1):32–37, 1992. ISSN 0736-6906. doi: 10.1145/142394.142401.

L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-19930-0.

L. J. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, pages 169–186, London, UK, 2002. Springer-Verlag. ISBN 3-540-43659-6.

S. Battista, F. Casalino, and C. Lande. MPEG-4: A multimedia standard for the third millennium, part 1. *IEEE MultiMedia*, 6(4):74–83, 1999. ISSN 1070-986X. doi: 10.1109/93.809236.

S. Battista, F. Casalino, and C. Lande. MPEG-4: A multimedia standard for the third millennium, part 2. *IEEE MultiMedia*, 7(1):76–84, 2000. ISSN 1070-986X. doi: 10.1109/93.839314.

J. Bennett. *Red Button Revolution - Power to the People*. 2004. URL http://www.bbc.co.uk/pressoffice/speeches/stories/bennett_mip.shtml.

T. Berners-Lee and M. Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999. ISBN 0062515861. Foreword By-Michael L. Dertouzos.

T. Berners-Lee, S. Hawke, and D. Connolly. Semantic web tutorial using n3. Turorial, 2004. URL http://www.w3.org/2000/10/swap/doc/.

E. Best and R. Devillers. Sequential and concurrent behaviour in Petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987. ISSN 0304-3975. doi: 10.1016/0304-3975(87)90090-9.

G. Blakowski and R. Steinmetz. A media synchronization survey: Reference model, specication, and case studies. *IEEE Journal on Selected Areas in Communications*, 14 (1):5–35, 1996.

T. Bodhuin, E. Guardabascio, and M. Tortorella. Migrating COBOL systems to the web by using the MVC design pattern. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 329, Washington, DC, USA, 2002. IEEE Computer Society.

B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988. ISSN 0018-9162. doi: 10.1109/2.59.

F. Boekhorst. Ambient intelligence, the next paradigm for consumer electronics:how will it affect silicon? In *IEEE International Solid-State Circuits Conference (ISSCC)*, pages 28–31, San Francisco, CA, USA, 2002. doi: 10.1109/ISSCC.2002.992922.

T. Bolognesi and E. Brinksma. Introduction to the ISO specification language lotos. *CComputer Networks and ISDN Systems*, 14(1):25–59, 1987. ISSN 0169-7552. doi: 10.1016/0169-7552(87)90085-7.

F. Bolton. *Pure CORBA*. Sams, 2001. ISBN 0672318121.

A. Bondarev. *Design of an Emotion Management System for a Home Robot*. Post-master thesis, Eindhoven University of Technology, 2002.

G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language for Object-Oriented Development (Version 0.9a Addendum)*. RATIONAL Software Corporation, 1996.

J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN 0-201-67494-7.

M. Broxvall and P. Jonsson. Towards a complete classification of tractability in point algebras for nonlinear time. In *Principles and Practice of Constraint Programming*, pages 129–143, 1999.

M. Broy. A logical basis for component-based systems engineering. In M. Broy and R. Steinbr uggen, editors, *Calculational System Design*. IOS Press, 1999.

B. Bruegge and A. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns, and Java*. Pearson Education, Inc. Pearson Prentice Hall, second edition, 2004.

M. C. Buchanan and P. T. Zellweger. Automatic temporal layout mechanisms. In *Computer Graphics (Multimedia '93 Proceedings)*, pages 341–350. Addison-Wesley, 1993.

M. C. Buchanan and P. T. Zellweger. Automatic temporal layout mechanisms revisited. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 1(1):60–88, 2005. ISSN 1551-6857. doi: http://doi.acm.org/10.1145/1047936.1047942.

M. C. Buchanan and P. T. Zellweger. Specifying temporal behavior in hypermedia documents. In *Proceedings of the ACM conference on Hypertext*, pages 262–271. ACM Press, 1992.

F. Buchmann and L. Bass. Introduction to the attribute driven design method. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 745–746, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

J. F. K. Buford. Architectures and issues for distributed multimedia systems. pages 45–64, 1994.

M. Bukowska. *Winky Dink Half a Century Later*. Post-master thesis, Eindhoven University of Technology, 2001.

D. C. A. Bulterman and L. Hardman. Structured multimedia authoring. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 1(1):89–109, 2005. ISSN 1551-6857. doi: 10.1145/1047936.1047943.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, Inc., 1996.

J. Cai, R. Kapila, and G. Pal. HMVC: The layered patterngamma+helmetal-desipattelemreus:95gamma+helmetal-desipattelemreus:95 for developing strong client tiers. *JavaWorld*, (July), 2000. URL http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.html.

G. Calvary, J. Coutaz, and L. Nigay. From single-user architectural design to PAC*: a generic software architecture model for cscw. In *CHI'97 Conference*, pages 242–249. Addison Wesley, 1997.

J. Cameron. The terminator, 1984. (DVD) by Cameron, James (Director), Daly, J. (Producer): MGM.

L. Carroll and C. B. Chorpenning. *Alice in Wonderland*. Dramatic Publishing Co., Woodstock, 1958. ISBN 0871290359.

T.-T. Chang, X. Wang, and Lim. Cross-cultural communication, media and learning processes in asynchronous learning networks. In *HICSS (the 35th Annual Hawaii International Conference on System Sciences)*, pages 113–122, 2002.

F. M. Cheung, K. Leung, J.-x. Zhang, H.-f. Sun, Y.-q. Gan, W.-z. Song, and D. Xie. Indigenous Chinese Personality Constructs: Is the Five-Factor Model Complete? *Journal of Cross-Cultural Psychology*, 32(4):407–433, 2001. URL http://jcc.sagepub.com/cgi/content/abstract/32/4/407.

H. F. Chua, J. E. Boland, and R. E. Nisbett. From The Cover: Cultural variation in eye movements during scene perception. *PNAS*, 102(35):12629–12633, 2005. URL http://www.pnas.org/cgi/content/abstract/102/35/12629.

J. A. Clapp. Rapid prototyping for risk management. In *the 11th International Computer Software and Applications Conference*, pages 17–22, Tokyo, Japan, 1987. IEEE Computer Society Press.

CMU. Aura project, 2005. URL http://www.cs.cmu.edu/$\sim$aura.

J. Cooper. *The Design Patterns java Companion*. Addision-Wesley Design Patterns Series, 1998. URL http://www.patterndepot.com/put/8/JavaPatterns.htm.

J. Coutaz. PAC, an implemention model for dialog design. In *Interact'87*, pages 431–436, Stuttgart, 1987.

J. Coutaz. PAC-ing the architecture of your user interface. In *4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, pages 15–32. Springer-Verlag, 1997.

M. Csikszentmihalyi. *Beyond Boredom and Anxiety: Experiencing Flow in Work and Play*. Jossey-Bass, 2000.

M. Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. HarperCollins, 1991.

D-BOX. *D-BOX Motion Simulators*. 2005. URL http://www.d-box.com.

A. M. Davis. Rapid prototyping using executable requirements specifications. *ACM SIGSOFT Software Engineering Notes*, 7(5):39–42, 1982.

M. de Graaf and L. Feijs. Support robots for playing games: the role of the player-actor relationships. In X. Faulkner, J. Finlay, and F. Detienne, editors, *People and Computers XVI - Memorable Yet Invisible*, pages 403–417. Springer-Verlag, 2002.

de Pinxi. Homepage of de Pinxi, 2003. URL http://www.depinxi.be/.

J. Derrick and E. Boiten. *Refinement in Z and object-Z: foundations and advanced applications*. Springer-Verlag, London, UK, 2001. ISBN 1-85233-245-X.

E. Diederiks. Buddies in a box - animated characters in consumer electronics. In W. L. Johnson, E. Andre, and J. Domingue, editors, *Intelligent User Interfaces*, pages 34–38, Miami, 2003. ACM Press.

J. Dong, J. Colton, and L. Zucconi. A formal object approach to real-time specification. In *the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*. IEEE Computer Society Press, 1996.

P. Dourish. *Where the Action Is: The Foundations of Embodied Interaction*. The MIT Press, 2001. ISBN 0262041960.

K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman. *Scenarios for Ambient Intelligence in 2010*. IST Advisory Group (ISTAG), 2001. URL ftp://ftp.cordis.lu/pub/ist/docs/istagscenarios2010.pdf.

D. Duce, M. Gomes, F. Hopgood, and J. Lee. *User Interface Management and Design*. Eurographic Seminar Series. Springer-Verlag, Berlin, 1991.

D. Duke, D. Duce, I. Herman, and G. Faconti. Specifying the PREMO synchronization objects. Technical Report ERCIM-01/97-RD48, ERCIM, 1997.

D. J. Duke, I. Herman, and M. S. Marshall. *PREMO: A Framework for Multimedia Middleware: Specification, Rationale, and Java Binding*, volume 1591 of *Lecture Notes in Computer Science*. Springer-Verlag Telos, 1999.

R. Duke and G. Rose. *Formal Object-oriented Specification using Object-Z*. Macmillan Press Limited, London, 2000. ISBN 0-333-80123-7.

G. Eddon and H. Eddon. *Inside COM+ Base Services*. Microsoft Programming Series. Microsoft Press, 2000. ISBN 0735607281.

W. K. Edwards. *Core JINI*. Prentice Hall PTR, 2nd edition, 2000. ISBN 0130894087.

B. Eggen, L. Feijs, M. Graaf, and P. Peters. Breaking the flow - intervention in computer game play through physical and on-screen interaction. In *the 'Level Up' Digital Games Research Conference*, 2003a.

B. Eggen, L. Feijs, and P. Peters. Linking physical and virtual interaction spaces. In *the International Conference on Entertainment Computing (ICEC)*, Pittsburgh, Pennsylvania, 2003b.

W. Eixelsberger and H. Gall. Describing software architectures by system structure and properties. In *COMPSAC '98 - 22nd International Computer Software and Applications Conference*, pages 106–111, Vienna, Austria, 1998. IEEE Computer Society.

R. Erfle. Hytime as the muitimedia document model of choice. In *the International Conference on Multimedia Computing and Systems*, pages 445–454, 1994.

L. Eronen and P. Vuorimaa. User interfaces for digital television: a navigator case study. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 276–279, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-252-2. doi: 10.1145/345513.345346.

Familiar. The Familiar project, 2005. URL http://familiar.handhelds.org.

M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997. ISSN 0001-0782.

L. Feijs and J. Hu. Component-wise mapping of media-needs to a distributed presentation environment. In *The 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 250–257, Hong Kong, China, 2004. IEEE Computer Society. ISBN 0-7695-2209-2. doi: 10.1109/CMPSAC.2004.1342840.

L. M. G. Feijs and Y. Qian. Component algebra. *Science of Computer Programming*, 42 (2–3):173–228, 2002.

J. Ferber. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman, 1999. ISBN 0-201-36048-9.

J. D. Foley and A. V. Dam. *Fundamentals of interactive computer graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982. ISBN 0-201-14468-9.

J. Forlizzi and S. Ford. The building blocks of experience: an early framework for interaction designers. In *Proceedings of the conference on Designing interactive systems*, pages 419–423. ACM Press, 2000.

A.-J. Fougeres. Agents to cooperate in distributed design. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 2629–2634, 2004. ISBN 0-7803-8566-7. doi: 10.1109/ICSMC.2004.1400727.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. ISBN 0321127420.

J. Freeman and J. Lessiter. Hear there & everywhere: The effects of multi-channel audio on presence. In J. Hiipakka, N. Zacharov, and T. Takala, editors, *ICAD 2001 - The Seventh International Conference on Audio Display*, pages 231–234. Helsinki University of Technology, 2001.

J. Freeman, J. Lessiter, and W. IJsselsteijn. An introduction to presence: A sense of being there in a mediated environment. *The Psychologist*, 14:190–194, 2001.

C.-L. Fung and M.-C. Pong. MOCS: An object-oriented programming model for multimedia object communication and synchronization. In *ICDCS*, pages 494–501, 1994.

Future TV. The future tv project, 1999. URL http://www.tml.tkk.fi/Research/future-tv/index.html.

A. Galton. A critical examination of allen's theory of action and time. *Artificial Intelligence*, 42(2–3):159–188, 1990.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

F. Gemperle, C. DiSalvo, J. Forlizzi, and W. Yonkers. The hug: A new form for communication. In *Designing the User Experience (DUX2003)*, New York, 2003. ACM Press.

GoF patterns. *Design Patterns Discussion (mailing list)*. 2005. URL http://mail.cs.uiuc.edu/mailman/listinfo/gang-of-4-patterns.

A. S. Gokhale and D. C. Schmidt. Measuring and optimizing CORBA latency and scalability over high-speed networks. *IEEE Trans. Comput.*, 47(4):391–413, 1998. ISSN 0018-9340. doi: 10.1109/12.675710.

C. F. Goldfarb. Hytime: a standard for structured hypermedia interchange. *Computer*, 24(8):81–84, 1991.

R. Gordon and S. Talley. *Essential JMF - Java Media Framework*. Prentice Hall, first edition, 1998.

M. Grand. *Patterns in Java, Volumn 1: A Catalog of Reusable Design Patterns Illustrated with UML*. Wiley Publishing, Inc., second edition, 2002.

W. Grosso. *Java RMI*. O'Reilly Media, Inc., 2001. ISBN 1565924525.

S.-U. Guan and S.-S. Lim. Modeling with enhanced prioritized Petri nets: Ep-nets. *Computer Communications*, 25(8):812–824, 2002.

S.-U. Guan, H.-Y. Yu, and J.-S. Yang. A prioritized Petri net model and its application in distributed multimedia systems. *IEEE Transactions on Computers*, 47(4):477–481, 1998. ISSN 0018-9340. doi: 10.1109/12.675716.

L. GuangChun, W. Lu, and X. Hanhong. A novel web application frame developed by MVC. *SIGSOFT Software Engineering Notes*, 28(2):7, 2003. ISSN 0163-5948. doi: 10.1145/638750.638779.

M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen. SOAP version 1.2 specification. W3C recommendation, 2003. URL http://www.w3.org/TR/soap.

K. Hafner. As robot pets and dolls multiply, 1999. URL http://www.teluq.uquebec.ca/psy2005/interac/robotsalive.htm.

L. Hardman, J. v. Ossenbruggen, K. S. Mullender, L. Rutledge, and D. C. A. Bulterman. Do you have the time? composition and linking in time-based hypermedia. In *Proceedings of the tenth ACM Conference on Hypertext and hypermedia : returning to our diverse roots*, pages 189–196. ACM Press, 1999.

H. R. Hartson and D. Hix. Human-computer interface development: concepts and systems for its management. *ACM Computing Surveys (CSUR)*, 21(1):5–92, 1989. ISSN 0360-0300. doi: 10.1145/62029.62031.

R. Harwig and E. Aarts. Ambient intelligence: invisible electronics emerging. In *IEEE 2002 International Interconnect Technology Conference*, pages 3–5, 2002.

P. Heckel. *The Elements of Friendly Software Design*, page 4. Sybex, 1991. ISBN 0895887681.

I. Herman, N. Correia, D. Duce, D. Duke, G. Reynolds, and J. Van Loo. A standard model for multimedia synchronization: PREMO synchronization objects. *Multimedia Systems*, 6(2):88–101, 1998.

L. Herrmann. Immersive broadcast: Concept and implementation. Technical Report Techinical Report C 2000 748, Philips LEP, 2000.

C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

M. E. Hodges, R. M. Sasnett, and M. S. Ackerman. Athena muse: A construction set for multimedia applications. *IEEE Software*, pages 37–43, 1989.

G. Hofstede. The cultural relativity of organizational practices and theories. *Journal of International Business Studies*, 14(2, Special Issue on Cross-Cultural Management): 75–89, 1983.

G. Hofstede. Cultural constraints in management theories. *Academy of management executive*, 7(1):81–94, 1993.

G. Hofstede and M. H. Bond. The confucius connection: From cultural roots to economic growth. *Organizational Dynamics*, 16(4):4–21, 1988.

J. Holmes. *Struts: The Complete Reference*. McGraw-Hill Osborne Media, 2004.

Honda. Asimo, 2002. URL http://www.honda.co.jp/ASIMO/.

J. Hoonhout. Development of a rating scale to determine the enjoyability of user interactions with consumer devices. Technical report, Philips Research, 2002.

T. Howard. *Smalltalk Developer's Guide to VisualWorks*. Cambridge University Press, 1995.

J. Hu. *Distributed Interfaces for a Time-based Media Application*. Post-master thesis, Eindhoven University of Technology, 2001.

J. Hu. Move, but right on time. In *1st European workshop on design and semantics of form and movement (DeSForM)*, pages 130–131, Newcastle upon Tyne, 2005. Philips. ISBN 0-9549587-1-3.

J. Hu. StoryML: Towards distributed interfaces for timed media. In W. ten Kate, editor, *Philips Conference InterWebT 2002*, NatLab, Eindhoven, 2002.

J. Hu. StoryML: Enabling distributed interfaces for interactive media. In *The Twelfth International World Wide Web Conference*, Budapest, Hungary, 2003. WWW. URL http://www2003.org/cdrom/.

J. Hu. Tony: Robotic toys for enriching media experience in home theathers. In *IST 2004 exhibition: When cognition meets design and technology - SOC's*, Amsterdam, 2004.

J. Hu and C. Bartneck. Culture matters - a study on presence in an interactive movie. In M. Slater, editor, *PRESENCE 2005, The 8th Annual International Workshop on Presence*, pages 153–159, London, UK, 2005. International Society for Presence Research, University College London. ISBN 0-9551232-0-8.

J. Hu and L. Feijs. An adaptive architecture for presenting interactive media onto distributed interfaces. In M. H. Hamza, editor, *The 21st IASTED International Conference on Applied Informatics (AI 2003)*, pages 899–904, Innsbruck, Austria, 2003a. ACTA Press. ISBN 0-88986-345-8.

J. Hu and L. Feijs. An agent-based architecture for distributed interfaces and timed media in a storytelling application. In *The 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, pages 1012–1013, Melbourne, Australia, 2003b. ACM. ISBN 1-58113-683-8. doi: 10.1145/860575.860771.

J. Hu, M. Janse, and H. Kong. User evaluation on a distributed interactive movie. In *HCI International 2005*, volume 3 - Human-Computer Interfaces: Concepts, New Ideas, Better Usability, and Applications, pages 735.1–10, Las Vegas, Nevada, USA, 2005. Lawrence Erlbaum Associates.

D. Hunter, A. Watt, J. Rafter, K. Cagle, J. Duckett, and B. Patterson. *Beginning XML (Programmer to Programmer)*. Wrox, 3rd edition, 2004. ISBN 0764570773.

A. Hussey. Using design patterns to derive PAC architectures from Object-Z specifications. In *Technology of Object-Oriented Languages and Systems (TOOLS 32)*, pages 40–51, 1999.

A. Hussey and D. Carrington. Comparing the MVC and PAC architectures: a formal perspective. *Software Engineering. IEE Proceedings*, 144:224– 236, 1997.

A. Hussey and D. Carrington. Using Object-Z to compare the MVC and PAC architectures. In C. Roast and J. Siddiqi, editors, *BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*. Springer, 1996.

IBC. The world of content creation, management and delivery, 2003. URL http://www.ibc.org.

IBM. Planet blue, 2005. URL http://www.research.ibm.com/compsci/planetblue.html.

ICE-CREAM. The ICE-CREAM project homepage, 2003. URL http://www.extra.research.philips.com/euprojects/icecream/.

Idealab. Robotics, robots, robot kits, oem solutions: Evolution robotics, 2003. URL http://www.evolution.com.

W. IJsselsteijn, H. Ridder, J. Freeman, and S. Avons. Presence: concept, determinants, and measurement. *Human Vision and Electronic Imaging V*, 3959 (1):520–529, 2000.

T. Illmann, M. Weber, A. Martens, and S. A. A pattern-oriented design of a web-based and case oriented multimedia training system in medicine. In *The 4th World Conference on Integrated Design and Process Technology*, Dallas, US, 2000.

In2Games. *Gametrak :: Come in and Play*. 2005. URL http://www.in2games.uk.com.

Insignia. Jeode java virtual machine, 2005. URL http://www.insignia.com/content/products/jvmProducts.shtml.

iRobot. iRobot, 2003. URL http://www.irobot.com.

ISO. Information processing - text and office systems - standard generalized markup language (sgml). Technical Report ISO 8879:1986(E), International Organization for Standardization, 1986.

ISO/IEC. Information technology - Z formal specification notation - syntax, type system and semantics. Technical Report 13568, ISO, 2002.

K. Ito. Ghost in the shell, 1992. (DVD) by Oshii, M. (Director), Ishikawa, M. (Producer): Palm Pictures/Manga Video.

I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997. ISBN 0-201-92476-5.

Jena. Jena – a semantic web framework for java, 2004. URL http://jena.sourceforge.net.

K. Jensen. *Coloured Petri nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1*. Springer-Verlag, London, UK, 1996. ISBN 3-540-60943-1.

W. Johnston. A type checker for Object-Z. Technical Report SVRC 96-24, Department of Computer Science, The University of Queensland, 1996.

P. Jordan. Human factors for pleasure in product use. *Applied Ergonomics*, 29(1): 25–33, 1998.

Kaffe. Kaffe java virtual machine, 2005. URL http://www.kaffe.org.

C.-M. Karat, C. Pinhanez, J. Karat, R. Arora, and J. Vergo. Less clicking, more watching: Results of the iterative design and evaluation of entertaining web experiences. In *8th International Conference on Human-Computer Interaction (INTERACT 2001)*, pages 455–463. IOS Press, 2001.

A. Khamis, D. Rivero, F. Rodriguez, and M. Salichs. Pattern-based architecture for building mobile robotics remote laboratories. In *IEEE International Conference on Robotics and Automation (ICRA'03)*, volume 3, pages 3284–3289, Taiwan, 2003a.

A. M. Khamis, F. J. Rodriguez, and M. A. Salichs. Remote interaction with mobile robots. *Autonomous Robots*, 15(3), 2003b. ISSN 0929-5593.

N. Khezami, S. Otmane, and M. Mallem. An approach to modelling collaborative teleoperation. In *12th International Conference on Advanced Robotics (ICAR)*, pages 788–795, Seattle, Washington, USA, 2005.

T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic. People, places, things: web presence for the real world. *Mobile Networks and Applications*, 7(5): 365–376, 2002. ISSN 1383-469X. doi: 10.1023/A:1016591616731.

A. Knight and N. Dai. Objects and the web. *IEEE Software*, 19(2):51–59, 2002. ISSN 0740-7459. doi: 10.1109/52.991332.

J. C. Knight and S. S. Brilliant. Preliminary evaluation of a formal approach to user interface specification. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 329–346, London, UK, 1997. Springer-Verlag. ISBN 3-540-62717-0.

R. Koenen. MPEG-4: Multimedia for our time. *IEEE Spectrum*, 36(2):26–33, 1999.

G. Krasner and S. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Program*, 1(3):26–49, 1988.

C. Kray, A. Krüger, and C. Endres. Some issues on presentations in intelligent environments. In E. Aarts, R. Collier, E. van Loenen, and B. de Ruyter, editors, *First European Symposium on Ambient Intelligence (EUSAI)*, pages 15–26, Veldhoven, The Netherlands, 2003. Springer.

D. Kruglinski. *Inside Visual C++*. Microsoft Press, 1995.

S. Kubrick. 2001: A space odyssey, 1968. (DVD) by Kubrick, S. (Director), Kubrick, S. (Producer): Warner Home Video.

P. Ladkin. *The Logic of Time Representation*. PhD thesis, University of California, 1987.

S. K. Langer. *Feeling and Form*. Prentice Hall, 1977.

K. A. Lantz, P. P. Tanner, C. Binding, K.-T. Huang, and A. Dwelly. Reference models, window systems, and concurrency. *ACM SIGGRAPH Computer Graphics*, 21(2): 87–97, 1987. ISSN 0097-8930. doi: http://doi.acm.org/10.1145/24919.24922.

B. Laurel. *Computers As Theatre*. Addison-Wesley Pub Co, 1993.

S. S. Laurent, E. Dumbill, and J. Johnston. *Programming Web Services with XML-RPC*. O'Reilly Internet Series. O'Reilly Media, Inc., 2001. ISBN 0596001193.

R. Lavender and D. Schmidt. Active object: an object behavioral pattern for concurrent programming. In J. M. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., 1996.

P. Le Hégaret and P. Schmitz. Java language binding (for synchronized multimedia integration language document object model). W3C recommendation, 2000. URL http://www.w3.org/TR/smil-boston-dom/java-binding.html.

LEGO. Lego.com mindstorms home, 2003. URL http://mindstorms.lego.com/eng.

J. Lessiter, J. Freeman, E. Keogh, and J. Davidoff. A cross-media presence questionnaire: The ITC sense of presence inventory. *Presence: Teleoperators and Virtual Environments*, 10(3):282–297, 2001.

N. Lévy and F. Losavio. Analyzing and comparing architectural styles. In *SCCC '99: Proceedings of the 19th International Conference of the Chilean Computer Science Society*, page 87, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0296-2.

N. Lévy, F. Losavio, and A. Matteo. Comparing architectural styles: broker specializes mediator. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 93–96, New York, NY, USA, 1998. ACM Press. ISBN 1-58113-081-3. doi: 10.1145/288408.288432.

D. Lightfoot. *Formal Specification Using Z*. Palgrave Macmillan, 2nd edition, 2001. ISBN 0333763270.

T. Little. Time-based media representation and delivery. In J. K. Buford, editor, *Multimedia Systems*, pages 175–200. ACM Press, 1995.

T. D. C. Little and A. Ghafoor. Interval-based conceptual models for time-dependent multimedia data. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):551–563, 1993. ISSN 1041-4347. doi: 10.1109/69.234768.

T. D. C. Little and A. Ghafoor. Multimedia synchronization protocols for broadband integrated services. *IEEE Journal on Selected Areas in Communications*, 9(9):1368–1382, 1991.

T. D. C. Little and A. Ghafoor. Synchronization and storage models for multimedia objects. *IEEE Journal on Selected Areas in Communications*, 8(3):413–427, 1990.

H. Liu and M. El Zarki. Delay and synchronization control middleware to support real-time multimedia services over wireless pcs networks. *IEEE Journal on Selected Areas in Communications*, 17(9):1660–1672, 1999. ISSN 0733-8716. doi: 10.1109/49.790488.

M. Lombart and T. Ditton. At the heart of it all: The concept of presence. *Journal of Computer Mediated Communication*, 3(2), 1997.

M. Lu and D. Walker. Do they look at educational multimedia differently than we do? a study of software evaluation in taiwan and the united states. *International Journal of Instructional Media*, 26(1), 1999.

Luqi and V. Berzins. Rapidly prototyping real-time systems. *IEEE Software*, 5(5):25–36, 1988. ISSN 0740-7459. doi: 10.1109/52.7941.

B. Mahony and J. Dong. Timed communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.

R. Mallart. Immersive broadcast reference application. White paper, Philips Research, 1999.

T. Malone and M. Lepper. *Making learning fun: A taxonomy of intrinsic motivations for learning*. Aptitude, learning and instruction. Volume 3: Cognitive and affective process analysis. Lawrence Erlbaum, 1987.

P. Markopoulos. *A compositional model for the formal specification of user interface software*. PhD thesis, Queen Mary and Westfield College, University of London, 1997.

P. Markopoulos, P. Johnson, and J. Rowson. Formal architectural abstractions for interactive software. *International Journal of Human-Computer Studies*, 49(5):675–715, 1998. ISSN 1071-5819. doi: 10.1006/ijhc.1998.0223.

P. Markopoulos, P. Shrubsole, and J. de Vet. Refinement of the PAC model for the component-based design and specification of television based interfaces. In D. Duke and A. Puerta, editors, *Design, Specification and Verification of Interactive Systems '99*, pages 117–132. Springer-Verlag, 1999.

B. McBride. Rdf primer. W3C recommendation, 2004. URL http://www.w3.org/TR/rdf-primer.

T. McComb. Refactoring Object-Z specifications. *Lecture Notes in Computer Science*, 2984:69–83, 2004. doi: 10.1007/b95935.

T. McComb and G. Smith. Architectural design in Object-Z. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, pages 77–86, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2089-8. doi: 10.1109/ASWEC.2004.1290460.

R. R. McCrae and O. P. John. An introduction to the five-factor model and its applications. *Journal of Personality*, 60(2):175–215, 1992.

D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

L. E. McKenzie and T. Snodgrass, Richard. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Computmg Surveys*, 23(4):501–543, 1991.

R. Menges, J. v.d. Asdonk, L. Scholten, and L. Admiraal. Mov'in: light, camera, action. Final project report DC222, Department of Industrial Design, Eindhoven University of Technology, Eindhoven, The Netherlands, 2005.

P. Menzel and F. D'Aluisio. *RoboSapiens*. MIT Press, Cambridge, 2000.

S. Metsker. *Design Patterns Java Workbook*. AddiAddison-Wesley, 2002.

T. Meyer, W. Effelsberg, and R. Steinmetz. A taxonomy on multimedia synchronization. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems*, pages 97–103, Lisbon, Portugal, 1993.

J. W. Michael Jeronimo. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press, 2003. ISBN 0971786119.

Microsoft. *Enterprise Solution Patterns Using Microsoft .Net: Version 2.0 : Patterns & Practices*. Microsoft Press, 2003.

MIT. Oxygen project, 2004. URL http://www.oxygen.lcs.mit.edu/.

J. Mueller. *Using SOAP*. Que, 2001. ISBN 0789725665.

Nasa. The mars exploration program, 2003. URL http://mars.jpl.nasa.gov.

NexTV. The NexTV project homepage, 2001. URL http://www.extra.research.philips.com/euprojects/nextv.

J. Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1993.

E. Niemelä and J. Marjeta. Dynamic configuration of distributed software components. In *ECOOP '98: Workshop ion on Object-Oriented Technology*, pages 149–150, London, UK, 1998. Springer-Verlag. ISBN 3-540-65460-7.

E. Niemela, J. Kalaoja, and P. Lago. Toward an architectural knowledge base for wireless service engineering. *IEEE Transaction on Software Engineering*, 31(5):361–379, 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.60.

L. Nigay and J. Coutaz. Building user interfaces: organizing software agents. In *Esprit'91 Conference Proceedings*, pages 707–719. ACM, 1991.

R. E. Nisbett and T. Masuda. Inaugural Articles: Culture and point of view. *PNAS*, 100 (19):11163–11170, 2003. URL http://www.pnas.org/cgi/content/abstract/100/19/11163.

R. Ogawa, H. Harada, and A. Kaneko. Scenario-based hypermedia: a model and a system. pages 38–51, 1992.

M. Okada. Muu: artificial creatures as an embodied interface. In *ACM Siggraph 2001*, page 91, New Orleans, 2001.

Y. Okada and Y. Tanaka. Collaborative environments of intelligentbox for distributed 3d graphic applications. In *CA '97: Proceedings of the Computer Animation*, page 22, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7984-0.

Y. Okada and Y. Tanaka. Intelligentbox: a constructive visual software development system for interactive 3d graphic applications. In *CA '95: Proceedings of the Computer Animation*, page 114, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7062-2.

OMG. *Event Service Specification Version 1.2*. Object Management Group, Inc., 2004a.

OMG. *Notification Service Specification Version 1.1*. Object Management Group, Inc., 2004b.

C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine. Evaluating policies and mechanisms to support distributed real-time applications with CORBA. *Concurrency and Computation: Practice and Experience*, 13(7):507–541, 2001. doi: 10.1002/cpe.558.

patterns-discussion. *General talk about software patterns (mailing list)*. 2005. URL http://mail.cs.uiuc.edu/mailman/listinfo/patterns-discussion.

D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992. ISSN 0163-5948.

A. Persidsky and M. Schaeffer. *Macromedia Director MX 2004 for Windows and Macintosh: Visual QuickStart Guide*. Visual QuickStart Guide. Peachpit Press, 2003. ISBN 0321193997.

C. A. Petri. *Kommunikation mit Automaten*. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

C. A. Petri. Kommunikation mit automaten. *New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377*, 1:1–Suppl. 1, 1966. English translation.

G. E. Pfaff, editor. *User Interface Management Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985. ISBN 038713803X.

Philips. iPronto online help, 2003. URL http://www.pronto.philips.com/products/ipronto/help/.

M. Pilgrim. *Dive Into Python*. Apress, 2004. ISBN 1590593561.

J. Pinnock. *Professional Dcom Application Development*. Wrox Press, 1998. ISBN 1861001312.

A. A. Poggio, J. J. G. L. Aceves, E. J. Craighill, D. B. Moran, L. Aguilar, D. R. L. Worthington, and J. M. Hight. CCWS: a computer-based, multimedia information system. *Computer*, 18(10):92–103, 1985. ISSN 0018-9162.

J. B. Postel, G. G. Finn, A. R. Katz, and J. K. Reynolds. An experimental multimedia mail system. *ACM Transactions on Information Systems (TOIS)*, 6(1):63–81, 1988. ISSN 1046-8188. doi: 10.1145/42279.42280.

B. Prabhakaran and S. V. Raghavan. Synchronization models for multimedia presentation with user participation. In *ACM Multimedia*, pages 157–166, 1993.

X. Qiu. Building desktop applications with web services in a message-based MVC paradigm. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 765, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2167-3. doi: 10.1109/ICWS.2004.31.

D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.01 specification. W3C recommendation, 1999. URL http://www.w3.org/TR/REC-html40.

R. Rajkumar, M. Gagliardi, and L. Sha. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *RTAS '95: Proceedings of the Real-Time Technology and Applications Symposium*, page 66, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-6980-2.

K. Ravindran and R. Steinmetz. Object-oriented communication structures for multimedia data transport. *IEEE Journal on Selected Areas in Communications*, 14 (7):1360–1375, 1996.

Regenbrecht and T. Schubert. Real and illusory interactions enhance presence in virtual environments. *Presence: Teleoperators and Virtual Environments*, 11(4):425–434, 2002.

J. K. Reynolds, G. G. Finn, J. B. Postel, A. L. DeSchon, and A. R. Katz. The DARPA experimental multimedia mail system. *Computer*, 18(10):82–89, 1985. ISSN 0018-9162.

Robodex. Robodex: Robot dream exposition, 2002. URL http://www.robodex.org.

RoboFesta. Robofesta (the international robot games festival), 2002. URL http://www.robofesta.net.

L. Rutledge. SMIL 2.0: XML for web multimedia. *IEEE Internet Computing*, 5(5): 78–84, 2001. ISSN 1089-7801. doi: 10.1109/4236.957898.

A. Sacau, J. Laarni, N. Ravaja, and T. Hartmann. The impact of personality factors on the experience of spatial presence. In M. Slater, editor, *The 8th International Workshop on Presence (Presence 2005)*, pages 143–151, London, 2005. International Society for Presence Research, University College London. ISBN 0-9551232-0-8.

C. Sas and G. M. P. O'Hare. Presence equation: an investigation into cognitive factors underlying presence. *Presence: Teleoper. Virtual Environ.*, 12(5):523–537, 2003. ISSN 1054-7460. doi: 10.1162/105474603322761315.

D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, first edition, 2000.

M. Schuemie, P. van der Straaten, M. Krijn, and C. van der Mast. Research on presence in VR: a survey. *ACM Virtual Reality Software and Technology (VRST)*, 4(2):335–341, 2001.

B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *AUUG97*, pages 243–255, Brisbane, Australia, 1997.

D. N. Serpanos and T. Bouloutas. Centralized versus distributed multimedia servers. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(8):1438–1449, 2000.

T. K. Shih, H.-C. Keh, L. Y. Deng, S.-E. Yeh, and C.-H. Huang. Extended timed Petri nets for distributed multimedia presentations. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 98, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-0990-8.

S. Shimojo, T. Matsuura, K. Fujikawa, S. Nishio, and H. Miyahara. A new hyperobject system harmony: its design and implementation. In *International conference on Multimedia information systems '91*, pages 243–260, New York, NY, USA, 1991. McGraw-Hill, Inc. ISBN 0-07-100660-5.

B. Shneiderman. *Designing the User Interface*. Addison Wesley, 3rd edition, 1997. ISBN 0201694972.

B. Shneiderman. Direct manipulation: A step beyond programming languages (abstract only). In *Proceedings of the joint conference on Easier and more productive use of computer systems. (Part - II)*, page 143, New York, NY, USA, 1981. ACM Press. ISBN 0-89791-064-8. doi: 10.1145/800276.810991.

Y. Shoham. Temporal logics in AI: Semantical and ontological considerations. *Artificial Intelligence*, 33(1):89–104, 1987.

A. J. H. Simons. The theory of classification, part 4: Object types and subtyping. *Journal of Object Technology*, 1(5):27–35, 2002.

M. Slater, V. Linakis, M. Usoh, and R. Kooper. Immersion, presence and performance in virtual environments: An experiment with tri-dimensional chess. *ACM Virtual Reality Software and Technology (VRST)*, pages 163–172, 1996.

G. Smith. Frequently asked questions, the Object-Z home page, 2005. URL http://www.itee.uq.edu.au/~smith/objectz.html.

G. Smith. *The Object-Z specification language*. Advances in formal methods. Kluwer Academic Publishers, 2000.

SoftWired AG. *iBus Programmer's Manual*. SoftWired AG, Zurich, Switzerland, 1998.

Sony. Aibo, 1999. URL http://www.aibo.com.

Sony. Sony develops small biped entertainment robot, 2002. URL http://news.sel.sony.com/pressrelease/2359.

J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *WICAS3: the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. ISBN 1-4020-7176-0.

J. Spivey. *The Z Notation: a reference manual*. Prentice Hall International (UK) Ltd, 2nd edition, 1992.

R. Srinivasan. *RPC: Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems, Inc., ONC Technologies, 1995.

S. Stelting and O. Maassen. *Applied Java Patterns*. Sun Microsystems Press, 2002.

C. Stephanidis. *User Interfaces for All: New perspectives into Human-Computer Interaction*, pages 3–17. Lawrence Erlbaum Associates, Mahwah, NJ, 2001. ISBN 0-8058-2967-9.

W. R. Stevens, B. Fenner, A. M. Rudoff, and R. W. Stevens. *Unix Network Programming, Vol. 1: The Sockets Networking API*. Addison-Wesley Professional, 3rd edition, 2003.

M. Stienstra. *Is every kid having fun? A gender approach to interactive toy design*. PhD thesis, Universiteit Twente, 2003.

R. E. K. Stirewalt and S. Rugaber. The model-composition problem in user-interface generation. *Automated Software Engineering*, 7(2):101–124, 2000. ISSN 0928-8910. doi: 10.1023/A:1008758107773.

M. Stokely and N. Clayton. *FreeBSD Handbook*. Dancing Goat Press, 2nd edition, 2001. ISBN 1571763031. URL http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/.

S. Sullivan, D. Brown, L. Winzeler, and S. Sullivan. *Programming With the Java Media Framework*. John Wiley & Sons, 1998.

Sun Microsystems. Java RMI specification. Technical report, 2003. URL http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

Sun Microsystems, Inc. *The InfoBus 1.2 Specification*. Sun Microsystems, Inc., 1999.

Tao. Intent java virtual machine, 2005. URL http://tao-group.com.

F. Tarpin-Bernard and B. David. AMF: a new design pattern for complex interactive software? In M. Smith, G. Salvendy, and K. R.J., editors, *Design of Computing Systems: social and ergonomic considerations. Proc. HCI International'97*, pages 351–354, San Francisco, 1997. Elsevier. ISBN 044482183X.

F. Tarpin-Bernard, B. T. David, and P. Primet. Frameworks and patterns for synchronous groupware: AMf-C approach. In *Proceedings of the IFIP TC2/TC13 WG2.7/WG13.4 Seventh Working Conference on Engineering for Human-Computer Interaction*, pages 225–241, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V. ISBN 0-412-83520-7.

O. Tezuka. Astro boy: The birth of astro boy, 2002. (VHS) by Ishiguro, N. and Tezuka, O. (Director), Yamamoto, S. (Producer): Manga Video.

G. Towner. *Discovering Quicktime: An Introduction for Windows and Macintosh Programmers*. Morgan Kaufmann, 1999.

H. C. Triandis. Cultural influences on personality. *Annual Review of Psychology*, 53 (1):133–160, 2002. URL http://arjournals.annualreviews.org/doi/abs/10.1146/annurev.psych.53.100901.135200.

D. Tsichritzis, S. Gibbs, and L. Dami. Active media. In D. Tsichritzis, editor, *Object Composition*, pages 115–132, Centre Universitaire d'Informatique, University of Geneva, 1991.

B. Ullmer and H. Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3-4):915–931, 2000. ISSN 0018-8670.

A. Ulrich and H. König. Specification-based testing of concurrent systems. In A. Togashi, T. Mizuno, N. Shiratori, and T. Higashino, editors, *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE 1997)*, pages 7–22. Chapman & Hall, 1997.

B. van Schooten. Modelling interaction in virtual environments using process algebra. In *15th Twente Workshop on Language Technology: Interactions in Virtual Worlds*, University of Twente, Enschede, The Netherlands, 1999.

B. van Schooten. Structuring distributed virtual environments using a relational database model. In *The Eighth Workshop on the Design, Specification and Verification of Interactive Systems*, Dept. of Computing Science, University of Glasgow, Scotland, 2001. URL http://www.dcs.gla.ac.uk/~johnson/papers/dsvis_2001/schooten/.

Velleman. K8000 PC interface board, 2002. URL http://www.velleman.be/ot/en/product/view/?id=9383.

J. Vlissides. Multicast. *C++ Report*, September, 1997a. URL http://www.research.ibm.com/designpatterns/pubs/multicast.html.

J. Vlissides. Multicast - Observer = Typed Message. *C++ Report*, November/December, 1997b. URL http://www.research.ibm.com/designpatterns/pubs/typed-msg.html.

A. Vogel, B. Kerhervé, G. von Bochmann, and J. Gecsei. Distributed multimedia and QOS: A survey. *IEEE MultiMedia*, 2(2):10–19, 1995. ISSN 1070-986X. doi: 10.1109/93.388195.

P. Vuorimaa. User interfaces for digital television: a navigator case study. In *IEEE International Conference on Multimedia and Expo*, volume 3, pages 1411–1414, New York, NY, USA, 2000. ISBN 0-7803-6536-4. doi: 10.1109/ICME.2000.871031.

A. Wachowski and L. Wachowski. The matrix, 1999. (DVD) by Wachowski, Andy and Wachowski, Larry (Director), Berman, B. (Producer): Warner Home Video.

T. Wahl and K. Rothermel. Representing time in multimedia systems. In *International Conference on Multimedia Computing and Systems*, pages 538–543, Boston, MA, 1994.

K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley Professional, second edition, 2004.

E. Waterworth, J. Waterworth, and L. R. The illusion of being present: Using the interactive tent to create immersive experiences. In *Presence 2001, The 4th Annual International Workshop on Presence*, 2001.

M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3): 94–104, 1991.

M. Weiser. Ubiquitous computing. *Computer*, 26(10):71–72, 1993.

R. Welch, T. Blackmon, A. Liu, B. Mellers, and L. Stark. The effects of pictorial realism, delay of visual feedback, and observer interactivity on the subjective sense of presence. *Presence: Teleoperators and Virtual Environments*, 5(3):263–273, 1996.

S. A. Wensveen. *A Tangibility Approach to Affective Interaction*. PhD thesis, Delft University of Technology, 2005.

G. J. Whitrow. *The Natural Philosophy of Time*. Clarendon Press, Oxford, second edition, 1980.

D. Winer. XML-RPC specification. Technical report, UserLand Software, Inc., 1999. URL http://www.xmlrpc.com/spec.

B. Witmer and M. Singer. Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, 1998.

J. Wojciechowski, B. Sakowicz, K. Dura, and A. Napieralski. MVC model, struts framework and file upload issues in web applications based on j2ee platform. In *International Conference Modern Problems of Radio Engineering, Telecommunications and Computer Science*, pages 342–345, 2004. ISBN 966-553-380-0. doi: 10.1109/TCSET.2004.1365980.

M. Woo, N. U. Qazi, and A. Ghafoor. A synchronization framework for communication of pre-orchestratedmultimedia information. *IEEE Network*, 8(1): 52–61, 1994. ISSN 0890-8044. doi: 10.1109/65.260079.

J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996. ISBN 0-13-948472-8.

C.-C. Yang. Design of the data-retrieving engine for distributed multimedia presentations. In *IEEE International Conference on Communications*, volume 10, pages 3237–3243, Helsinki, Finland, 2001. ISBN 0-7803-7097-1. doi: 10.1109/ICC.2001.937268.

C.-C. Yang and J.-H. Huang. A multimedia synchronization model and its implementation in transport protocols. *IEEE Journal on Selected Areas in Communications*, 14(1):212–225, 1996.

C.-C. Yang, C.-W. Tien, and Y.-C. Wang. Modeling of the non-deterministic synchronization behaviors in SMIL 2.0 documents. In *2003 International Conference on Multimedia and Expo (ICME'03)*, volume 3, pages 265–268, 2003. ISBN 0-7803-7965-9.

F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (third edition), 2004. URL http://www.w3.org/TR/2004/REC-xml-20040204/. W3C Recommendation.

W. Zhao and D. Kearney. Deriving architectures of web-based applications. *Lecture Notes in Computer Science*, 2642:301–312, 2003.

Publications related to this research

Wen Xu, Karel Kreijns, and Jun Hu. Designing social navigation for a virtual community of practice. *Edutainment: Technology and Application, Lecture Notes in Computer Science*, 3942:27–38, 2006.

M.D. Janse, P. van der Stok, and J. Hu. Distributing multimedia elements to multiple networked devices. In *User Experience Design for Pervasive Computing, Pervasive 2005*, Munich, Germany, 2005. URL http://www.fluidum.org/events/experience05/.

J. Hu. Move, but right on time. In *1st European workshop on design and semantics of form and movement (DeSForM)*, pages 130–131, Newcastle upon Tyne, 2005. Philips. ISBN 0-9549587-1-3.

J. Hu, M.D. Janse, and H. Kong. User evaluation on a distributed interactive movie. In *HCI International 2005*, volume 3 - Human-Computer Interfaces: Concepts, New Ideas, Better Usability, and Applications, pages 735.1–10, Las Vegas, Nevada, USA, 2005. Lawrence Erlbaum Associates.

J. Hu and C. Bartneck. Culture matters - a study on presence in an interactive movie. In Mel Slater, editor, *PRESENCE 2005, The 8th Annual International Workshop on Presence*, pages 153–159, London, UK, 2005. International Society for Presence Research, University College London. ISBN 0-9551232-0-8.

C. Bartneck and J. Hu. Presence in a distributed media environment. In *User Experience Design for Pervasive Computing, Pervasive 2005*, Munich, Germany, 2005. URL http://www.fluidum.org/events/experience05/.

M.D. Janse (ed.) and all partners. The ICE-CREAM project: Final report. Technical Report IST-2000-28298 ICE-CREAM-phr-0404-01/Janse, IST, 2004. URL http://www.hitech-projects.com/euprojects/icecream/deliverables.htm.

J. Hu. Tony: Robotic toys for enriching media experience in home theathers. In *IST 2004 exhibition: When cognition meets design and technology - SOC's*, Amsterdam, 2004.

L.M.G. Feijs and J. Hu. Component-wise mapping of media-needs to a distributed presentation environment. In *The 28th Annual International Computer Software and Applications Conference (COMPSAC 2004)*, pages 250–257, Hong Kong, China, 2004. IEEE Computer Society. ISBN 0-7695-2209-2. doi: 10.1109/CMPSAC.2004.1342840.

C. Bartneck and J. Hu. Rapid prototyping for interactive robots. In F. Groen, N. Amato, A. Bonarini, E. Yoshida, and B. Kröse, editors, *The 8th Conference on Intelligent Autonomous Systems (IAS-8)*, pages 136–145, Amsterdam, 2004. IOS press.

J. Hu and L.M.G. Feijs. An adaptive architecture for presenting interactive media onto distributed interfaces. In M. H. Hamza, editor, *The 21st IASTED International Conference on Applied Informatics (AI 2003)*, pages 899–904, Innsbruck, Austria, 2003a. ACTA Press. ISBN 0-88986-345-8.

J. Hu and L.M.G. Feijs. An agent-based architecture for distributed interfaces and timed media in a storytelling application. In *The 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, pages 1012–1013, Melbourne, Australia, 2003b. ACM. ISBN 1-58113-683-8. doi: 10.1145/860575.860771.

J. Hu. StoryML: Enabling distributed interfaces for interactive media. In *The Twelfth International World Wide Web Conference*, Budapest, Hungary, 2003. WWW. URL http://www2003.org/cdrom/.

C. Fuhrhop, J. Hu, and O. Gafsou. Prototypes of applications, user interfaces ad end user tools. Technical report, ICE-CREAM (IST-2000-28298), 2003.

M.D. Janse (ed.) and all partners. The NexTV project: Final report. Technical Report IST-1999-11288 NexTV-Philips-0204-67/Janse, IST, 2002. URL http://www.hitech-projects.com/euprojects/nextv/deliver/public.htm.

J. Hu. StoryML: Towards distributed interfaces for timed media. In W. ten Kate, editor, *Philips Conference InterWebT 2002*, NatLab, Eindhoven, 2002a.

J. Hu. Enabling distributed interfaces for immersive storytelling (poster presentation). In *Interaction Design and Children 2002*, TU/e, Eindhoven, 2002b.

J. Hu. *Distributed Interfaces for a Time-based Media Application*. Post-master thesis, Eindhoven University of Technology, 2001.

Summary

The concept of Ambient Intelligence (AmI) is introduced by Philips Research as a new paradigm in how people interact with technology. It envisions digital environments to be sensitive, adaptive, and responsive to the presence of people. This PhD project believes that AmI environments will change the way people use multimedia services. The environments, which include many devices, will play interactive multimedia to engage people in a more immersive experience than just watching television shows or listening to radio programs. People will interact not only with the environment itself, but also with the interactive multimedia through the environment.

For many years, the research and development of multimedia technologies have increasingly focused on models for distributed applications, but the focus was mainly on the distribution of the media sources. Within the context of AmI, not only are the media sources distributed, the presentation of and the interaction with the media will also be distributed across interface devices. This PhD project focuses on the design of the structure of multimedia content and the distributed interfaces, believing that the user experience of multimedia in a distributed environment can be enriched by structuring both the media content at the production side and the playback system architecture at the user side in a proper way. The structure should enable both the media presentation and the user interaction to be distributed and synchronized over the networked devices in the environment. The presentation and interaction should be adaptive to the profiles and preferences of the users, and the dynamic configurations of the environment.

The design process went through three design iterations, following a spiral model. The first iteration was needed to get some first-hand experience and the preliminary requirements. It was concluded with the structure of StoryML and a demonstrator TOONS, which put the requirements for the second iteration on a stable foundation. More user requirements and technical challenges emerged in the second iteration during the development of the demonstrator DeepSea. DeepSea was used for the user evaluation of the concept of interactive multimedia in distributed environments. The design and development were brought forward based on the technical requirements and the experience gained from the second iteration. The design was completed with a full implementation of the proposed architecture, based on open standards and technologies. Three more demonstrators (TOONS in IPML, TheInterview, Mov'in) were built to validate the design. TheInterview was used to test the effect of the user's cultural background on their perception of presence in interacting with these multimedia presentations in a distributed setting. During the project, the proposed architecture was used in implementing applications for various projects, ranging from a big EU project together with professional developers, to a small educational project by a team of four second year university students. Each resulted in a working demonstrator or prototype.

During the design process, several formal methods were used, including Object-Z, Broy's component based framework, and the Petri net based OCPN and ASE. The formal approaches were helpful: They are abstract enough to make mechanisms explicit in a clear way and at the same time they are concrete enough to see the structure of the implementation. However it was not possible to formalize every aspect of the system. The system is far too complex for that. Several formalisms and several types of "syntactic sugar" were used to keep things manageable. Yet the most important design concepts were covered by formalizing them in a pragmatic way.

As a direct result of this design, a generic architecture has been implemented on top of existing network technologies and platform architectures, to enable playing IPML scripts in a networked environment with user preference and dynamic configurations taken into account. Aside from the architecture, this project also contributed to the body of scientific and engineering results. Examples are the method of rapid robotic prototyping, the extension of Petri nets as action synchronization engine, and new architectural patterns such as Timed Action, Synchronizable Object, Real-time Channel and Streaming Channel.

The results of user evaluations offer preliminary insight into how distribution, level of control and number of users influence user's fun and presence experience in interacting with an AmI movie. Notably the level of control, and the number of the users, have a positive influence on the user experience. The influence of distribution on the user experiences is also observed and there can be an effect depending on the type of the distributed content and how the distribution is arranged. An effect of cultural background upon the perception of presence in a distributed setting is also observed. As far as known, it is the first time that the long-standing conjecture on this effect is confirmed experimentally.

Samenvatting

Het concept van Ambient Intelligence (AmI) is geïntroduceerd door Philips Research als een nieuw paradigma voor interactie tussen mens en technologie. In deze visie worden omgevingen met digitale technologie sensitief, adaptief en responsief met betrekking tot de aanwezigheid van mensen. Dit promotieproject gaat uit van de overtuiging dat AmI-omgevingen veranderingen teweeg zullen brengen in de manier waarop mensen multimedia-diensten gebruiken. De omgevingen, waarin veel apparaten een rol spelen, zullen interactieve multimedia afspelen om mensen te betrekken in een ervaring die veel meer een totaalervaring is dan gewoon kijken naar televisieshows of luisteren naar radioprogramma's. Mensen hebben niet alleen interactie met de omgeving zelf, maar via die omgeving ook met de interactieve multimedia.

In de loop der jaren zijn het onderzoek en de ontwikkeling van multimedia technologieën steeds meer gericht op modellen voor gedistribueerde toepassingen, maar de nadruk lag vooral op de distributie van de mediabronnen. In de context van AmI zijn niet alleen de mediabronnen gedistribueerd, maar zullen ook de presentatie van en de interactie met de media over interface-apparaten gedistribueerd zijn. Dit promotieproject richt zich op het ontwerp van de structuur van de multimedia-inhoud en de gedistribueerde interfaces, vanuit de overtuiging dat de gebruikerservaring van multimedia in een gedistribueerde omgeving verrijkt kan worden door zowel de media-inhoud aan de productiezijde als het afspeelsysteem aan de gebruikerszijde op een goede manier te structureren. Deze structuur moet mogelijk maken dat zowel de mediapresentatie als de gebruikersinterface worden gesynchroniseerd en gedistribueerd over de genetwerkte apparaten in de omgeving. De presentatie en interactie moeten adaptief zijn met betrekking tot de profielen en de voorkeuren van de gebruikers, alsmede met betrekking tot de dynamische configuraties van de omgeving.

In het ontwerpproces zijn drie ontwerpiteraties doorlopen volgens een spiraalmodel. De eerste iteratie was nodig om eerste ervaringen op te doen en om een voorlopig programma van eisen te verkrijgen. Ze werd afgesloten met de structuur van StoryML en een demonstratie opstelling TOONS, hetgeen een degelijke basis leverde voor het programma van eisen van de tweede iteratie. Aanvullende gebruikerseisen en technische uitdagingen kwamen tevoorschijn tijdens de tweede iteratie bij de ontwikkeling van de DeepSea-demonstratie-opstelling. DeepSea is gebruikt voor de gebruikersevaluatie van het concept van interactieve multimedia in gedistribueerde omgevingen. Het ontwerp en de ontwikkeling borduurden voort op de technische eisen en de ervaringen van deze tweede iteratie. Het ontwerp is afgerond met een volledige implementatie van de voorgestelde architectuur, die geheel gebaseerd is op open standaarden en technologieën. Drie extra demonstatie-opstellingen (TOONS in IPML, TheInterview, Mov'in) zijn gebouwd om het ontwerp te valideren. TheInterview is gebruikt om het effect te onderzoeken van de culturele achtergrond van de gebruikers met betrekking tot hun waarneming van 'presence' tijdens de interactie met de multimediapresentie in een gedistribueerde opzet. Tijdens het project is de voorgestelde architectuur gebruikt om toepassingen voor verschillende projecten te implementeren, variërend van een groot EU-project met professionele ontwikkelaars, tot een klein educatief project door een groep van vier tweedejaars universiteitsstudenten. Elk resulteerde in een werkend demonstratiemodel of prototype.

Tijdens het ontwerptraject zijn verscheidene formele methoden gebruikt, inclusief Object-Z, het op componenten gebaseerde raamwerk van Broy, en de op Petrinetten gebaseerde OCPN en ASE. De formele methoden bleken nuttig: enerzijds zijn ze abstract genoeg om bepaalde mechanismen op een duidelijke manier te expliciteren, anderzijds zijn ze voldoende concreet om de structuur van de implementatie nog te kunnen zien. Het was echter niet haalbaar om elk aspect van het systeem te formaliseren. Daarvoor is het systeem veel te complex. Verscheidene formalismen en verscheidene soorten 'syntactische suiker' zijn gebruikt teneinde een en ander hanteerbaar te houden. Nochtans zijn de voornaamste ontwerpconcepten afgedekt door ze op een pragmatische manier te formaliseren.

Als direct resultaat van dit ontwerp is een generieke architectuur gerealiseerd bovenop bestaande netwerktechnologieën en platformen, die het mogelijk maakt om IPML scripts af te spelen in een genetwerkte omgeving, rekening houdend met de voorkeuren van de gebruikers en met dynamische configuraties. Naast de genoemde architectuur heeft het project ook bijgedragen aan de stand van wetenschappelijke kennis en techniek. Voorbeelden zijn de methode van 'rapid robotic prototyping', de uitbreidingen van Petri-netten tot 'action synchronisation engine', en een aantal nieuwe architectuurpatronen zoals Timed Action, Synchronizable Object, Real-time Channel en Streaming Channel.

Daarnaast verschaffen de gebruikersevaluaties voorlopige inzichten hoe de mate van distributie, de mate van 'control' en het aantal gebruikers van invloed zijn op het plezier en presence-beleving bij een AmI-film. Met name de mate van control en het aantal gebruikers hebben een positieve invloed op de beleving van de gebruikers. De invloed van distributie op de beleving van de gebruikers is ook waargenomen en er kan een effect zijn, afhankelijk van de aard van de gedistribueerde inhoud en hoe de distributie in elkaar steekt. Ook is een effect van culturele achtergrond waargenomen op de perceptie van presence in een gedistribueerde opzet. Voorzover bekend is het de eerste keer dat deze reeds lang bestaande hypothese experimenteel bevestigd wordt.

Curriculum Vitae

Jun Hu is born on the 11th of February (Lunar calendar), 1969, in Jiangsu, China. In 1990, He graduated cum laude from Northwest University (Xi'an) with a Bachelor of Science degree in Computational Mathematics. He was then working on digital signal processing, information management systems and computer aided design systems, for computational centers of an oil exploration company in Nanjing and a construction machinery company in Xi'an, with qualified titles of Senior Programmer and later System Analyst.

While working, he started his study again at Northwest University in 1996. He graduated cum laude and received a Master of Engineering degree in Computer Science in 1999. His dissertation *Content-based Retrieval of a Medical Image Database* was awarded Outstanding Master Thesis by the university.

After some journeys he then moved to Holland and joined a 2-year postmaster program in User-system Interaction at Eindhoven University of Technology. In 2001 he finished the project *Distributed Interfaces for a Time-based Media Application* at Philips Research and received a Professional Doctorate in Engineering degree.

He continued the research and design of distributed architectures for interactive media as a PhD project, at the department of Industrial Design of Eindhoven University of Technology, in cooperation with Philips Research. Since 2003, he has been a teacher at this department, where the students are determined to design "intelligent products and services".

Background materials

BACKGROUND MATERIAL A

Rapid Robotic Prototyping[1]

An increasing number of robots are being developed to directly interact with humans. This can only be achieved by leaving the laboratories and introduce the robots to the real world of the users. This may be at home, work or any other location that users reside. Interactive robots are already commercially available, such as Aibo (Sony, 1999), SDR (Sony, 2002) and the products of the iRobot company (iRobot, 2003). Their applications range from entertainment to helping the elderly (Gemperle, DiSalvo, Forlizzi, and Yonkers, 2003), operation in hazardous environments (iRobot, 2003) and especially for this project, interfaces agents for AmI homes (Aarts et al., 2001).

A major problem for the development of such robots is the definition of user requirements. Since the users have usually no prior experience it is impossible to simply interview them on how they would like their robot to be. To overcome this problem a rapid robotic prototyping method has been used that directly relates to the well-known rapid software prototyping method.

A.1 Problem Definition

Humans have generally very limited experience interacting with robots. Their experiences and expectations are usually based on movies and books and therefore cultural dependent. The great success of robotic show events, such as RoboFesta (2002) and Robodex (2002), show that Japanese have a vivid interest in robots and consider them as partners to humans. Their positive attitude may be based on years of Anime cartoons, starting in the fifties with “Astro Boy” (Tezuka, 2002) and later in “Ghost in the Shell” (Ito, 1992) in which robots save humanity from various threats. In comparison, the attitude of Europeans is less positive. The success of movies such as “2001 Space Odyssey” (Kubrick, 1968), “Terminator” (Cameron, 1984) and “The

[1]This part of work is revised from a paper written together with Christoph Bartneck, published in the proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS-8) (Bartneck and Hu, 2004).

Matrix" (Wachowski and Wachowski, 1999) shows a deep mistrust towards robots. The underlying fear is that robots might take over control and enslave humanity.

Against such a cultural load, the appearance of robots is of major importance. It determines the attitude and expectations towards it. If, for example, it has a very human form people are likely to start talking to it and would expect it to answer. If the robot cannot comply with these expectations, the user will have a disappointing experience (Diederiks, 2003).

The problem lies in the exact definition of how such a robot should look and how it should behave. It is not possible to draw these requirements directly from users by interviewing them, since they have no prior experience and are largely influenced by culture as mentioned above. These difficulties should not tempt developers to ignore these requirements. Too often complex and expensive robots are developed without these requirements in mind (Bondarev, 2002). Once such a robot is finished and showed to users it often conflicts with the user's needs and expectations and therefore does not gain acceptance.

Typical challenges in the development process of robots are discussed first, then a method to tackle them is proposed.

A.1.1 Design challenges

Shape To reduce development costs many parts of a robot are based on standard components. They are usually stacked on top of each other and once it is operational a shell is build around it to hide it from the user. The shape of the robot is only considered after the technology is built (Bondarev, 2002). Humans are very sensitive to proportions of anthropomorphic forms and therefore these robots are often perceived as mutants due to their odd shapes. The evolution of Honda's Asimo (Honda, 2002) is a positive example of integrating technology into a natural shape .

Purpose Building robots is a challenging and exciting activity and some engineers build robots only for the fun of it. However, a robot in itself is senseless without a purpose. A clear definition of its purpose is necessary to deduct requirements which in turn increase the chance of the robot to become a success. An unfortunate example of a wrong purpose is Kuma (Hafner, 1999). It was intended to reduce the loneliness of elderly people by accompanying them during watching Television. It is unclear how that would increase human contact and hence tackle the root of the problem. A robot that improves communication (Gemperle et al., 2003) would be more successful.

Social role The robot will show intentional behavior and therefore humans will perceive the robot to have a character (Bartneck, 2002). Together with its purpose the robot plays a social role, for example the one of a butler. Such a role entails certain expectations. For example, you would expect a butler to be able to serve you drinks *and* food. A robot that would only be able to do the one and not the other would lead to a disappointing experience.

Environment To be able to define the purpose and role of the robot it is necessary to consider its environment. What are the characteristics of it in terms of architectural and social structures? Sony's Aibo, for example, is not able to overcome even a small step and therefore its action range is much more limited than the one of every dog. This results in some frustration of its owners (Menzel and D'Aluisio, 2000).

Technical challenges Although people have different ideas of what constitutes a robot, most of the specialists agree that robots should at least have a programmable brain, a number of actuators that affect the environment, and possibly some sensors keep tracking the environment. The robot's computer or processor as the brain controls the actuators, such as electric motors, solenoids, and hydraulic or pneumatic systems, to make physical movements. Many robots have sensory systems, and few have the ability to see, hear, smell or taste. The most common robotic sense is the sense of movement - the robot's ability to monitor its own motion. These are the basic nuts and bolts of robotics. Specialists can combine these elements in an infinite number of ways to create robots of unlimited complexity.

Building real robots is hard. Robots are complex systems which rely on software, electronics and mechanical systems all working together. It requires the specialists to have the knowledge and skills that cover all these disciplines to bring a robot alive. Building robots such as the NASA's Mars Pathfinder (Nasa, 2003) and the Honda's Asimo (Honda, 2002) is a mission impossible for an individual. A team of top scientists and engineers from different backgrounds need to work together closely in order to build such robots. These robots are therefore very expensive. Small robots such as Sony's Aibo are not cheap either. Although it is designed for home entertainment and many families can afford one of the Aibo, Sony must have paid a fortune for the designers, researchers and engineers in order to bring such a not-so-expensive product into the market. Most robots to date have been more like kitchen appliances. Specialists build them from the ground up for a fairly specific purpose. They cost a lot but do not adapt well to radically new applications.

There are also obstacles that lie in the software design, specifically for human-like robots. People may expect such robots to talk and think only because they look like they can. Technologies such as speech recognition are still at a level of pattern matching of sound signals and not yet mature enough to *understand* the conversations. Artificial Intelligence may help the robot to understand, but AI itself is still largely theoretical. AI is arguably the most exciting realm in robotics and has successfully made computers to solve certain problems and learn in a limited capacity but still many challenges remain. Those who expect off-the-shelf solutions from the AI often get disappointed.

Empirical evaluation Another challenge in the development of interactive robots is the definition of measurable benefits. These benefits are influenced by the interaction with humans and hence difficult to measure. A wide range of methods and measurement tools are available from the human computer interaction research area (Nielsen, 1993) to help with the evaluation. Here only some of the common challenges that developers experience in the evaluation process are described.

The first is of course not to do any evaluation. Simply stating that a certain robot is fun to interact with is nothing more than propaganda. A first difficulty in the evaluation process is to clearly define what the actual benefit of the robot should be, how it can be measured and who the target user group is. Especially for vague concepts, such as “fun”, it is difficult to find validated measurement tools. Furthermore, too few participants that are possibly even colleagues of the developers also often compromise the tests. A small and technical oriented group of participants does not allow a generalization across the target group. The participants need to come from the group of intended users. A problem with these users is the novelty effect (Bartneck, 2002; Diederiks, 2003). Interacting with a robot is exciting for users that have never done it before and hence their evaluation tends to be too positive.

A.2 Solution

Similar problems and difficulties as described above exist also in software engineering. The similarity of the problems and difficulties lie in the software engineering suggests that the principles of rapid prototyping may also be applicable in the development of interactive robots.

A.2.1 Rapid prototyping in software engineering

Requirements definition is crucial for a successful software development, but it is often found to be difficult if only interviewing the users. Some users either have expectations for computers that is high enough to lead requirements to more than what is really needed, or disbelieve computers can solve certain problems thus unconsciously hide the requirements from the developers. With limited prior experience of existing software solutions, very often the users can not specify precisely the requirements until they experience some of the solutions. Lacking of the user’s domain knowledge, the software developers also often find it hard to explain to the users what is feasible until they manage to visualize the ideas.

To tackle the problem, the rapid prototyping model is introduced against the framework of the conventional water-fall life cycle models (Clapp, 1987; Davis, 1982; Luqi and Berzins, 1988). The rapid prototyping model strives for demonstrating functionality early in the development of a software development, in order to draw requirements and specifications. The prototype provides a vehicle for the developers to better understand the environment and the requirements problem being addressed. By demonstrating what is functionally feasible and where the technical weak spots still exist, the prototype stretches their imagination, leading to more creative and realistic inputs, and a more forward-looking system.

Not only does rapid prototyping help to increase the software correctness with regard to its functionality, but also it improves its usability by involving the end users in the early development (Nielsen, 1993). It provents the developers from focusing only on the functionality then later struggling to find a usable wrapping outfit for the end users. The development of the final system can be much faster and much cheaper by involving early usability evaluation based on the prototypes.

The final software system should not be an isolated thing. It has to fit into certain hardware and software environments, and has to have a proper role in the user’s social activities (Laurel, 1993). Rapid prototyping provides a test bed for the developers to study where the fence line and the gates of the system should be, and how well it can live along with the neighborhood. By experiencing with the prototype, hence with a better understanding how the human-computer activities could be redefined and reallocated, the users may provide valuable requirements and suggestions for a clearer definition of the role of the system in these activities that fits the user's expectations.

Before elaborating the details of rapid robotic prototyping, let's first have a look at what makes it different from software prototyping and why it is worth a discussion.

A.2.2 Difference between robotic prototyping and software prototyping

The first difference is that the target system of robotic prototyping is a robot, which has its physical existence in the 3D world, while a software system is just an artifact that exists digitally in a virtual space. One of the most important goals of rapid robotic prototyping is to investigate the user requirements of the physical appearance and behavior, hence implementing a robotic prototype is not just programming to give it intelligence, but more importantly, to build its physical embodiments. Software prototyping often uses existing software packages, modules and components to accelerate the process, while robotic prototyping often needs electronic and mechanical building blocks.

The physical embodiments of the interactive robots encourage naturally the tactile human-robot interaction. The user and the robot exchange the tactile information, ranging from force, texture, gestures to surface temperature. The tactile interaction is seldom necessary in most of the software systems, the interfaces of which are often confined in a 2D screen. The 2D interfaces of such could be easily prototyped using low-fidelity techniques such as paper mock-ups, computer graphics and animations, whereas these techniques are not suitable for prototyping the tactile interaction that is essential to most of the interactive robotic systems.

Another difference lies in the intermediate prototype. The prototypes built during the software prototyping can possibly be evolving or growing into a full functioning system. This is so called evolutionary prototyping. During the process of evolution, a software design emerges from the prototypes. Rapid robotic prototyping proposed here is only for quickly eliciting the user requirements. To make the process faster, the efficiency, reliability, intelligence and the building material of the robot is less of importance. Hence the prototypes produced in robotic prototyping are not intended to be ready for industrial reproduction.

A.2.3 Rapid robotic prototyping techniques

Keeping these differences in mind, it is now ready to review the often used prototyping techniques and propose the corresponding methods in robotic prototyping, following two dimensions of robotic prototyping: Horizontal prototyping realizes the appearance but eliminates depth of the behavior implementation, and vertical prototyping gives full implementation of certain selected behaviors. If the focus is on the appearance or the interface part of the robot, horizontal prototyping is needed and it results in a surface layer that includes the entire user interface to a full-featured robot but with no underlying functionality; if prototyping is to explore the details of certain features of the robot, vertical prototyping is necessary in order to be tested in depth under realistic circumstances with real user tasks.

Scenarios Without any horizontal and vertical implementation, scenarios are the minimalist, and possibly the easiest and cheapest prototype in which only a single interaction session is described, encapsulating a story of a user interacting with robotic facilities to achieve a specific outcome under certain circumstances over a time interval. As in software prototyping, scenarios can be used during the early requirement analysis to inspire the user's imagination and feedback without the expense of constructing a running prototype. The form of the prototype can be a written narrative, or detailed with pictures, or even more detailed with video.

Paper mock-ups In software prototyping, paper mock-ups are usually based on the drawings or printouts of the 2D interface objects such as menus, buttons, dialog boxes and their layout. They are turned into functioning prototypes by having a human "play computer" and present the change of interface whenever the user indicates an action. The system is horizontally mocked up with low fidelity techniques, vertically faked with human intelligence.

Although they are also very useful in robotic prototyping to make the scenarios interactive, paper mock-ups in robotic prototyping are of even lower fidelity. In software prototyping, 2D interfaces are mocked up with the 2D objects on paper. To reach the same fidelity level in robotic prototyping, the 3D robot should be mocked-up with a box of sculptures or 3D "print-outs" instead of a pile of drawings, which is time-consuming and expensive, though technically possible. Prototyping the 3D robotic appearance and behavior on 2D paper is just like prototyping the 2D software interface on a 1D line. Too much fidelity is lost. This argument has led us to mock-up the robots using robotic kits.

Mechanical Mock-ups To keep the horizontal fidelity to a certain level, it is necessary to mock up the physical 3D appearance and mechanical structure of the robot. Robotic kits such as Evolution Robotics (Idealab, 2003) and LEGO Mindstorms (LEGO, 2003) are good tools to build such mock-ups. These kits come with not only common robotics hardware such as touch sensors, rotation sensors, temperature sensors, step motors and video cameras, but also mechanical parts such as beams, connectors and wheels, and even ready made robot body pieces and joints. One can assemble a robot easily and quickly according to the needs, and can expect less workload of mechanical and electronic design.

To make a mock-up, only mechanical parts are needed to build up the skeleton. With some simple clothing, the robot appearance can be built with a higher fidelity. The behavior of the robot can be faked up by a human manipulating the prototype like a puppet show, according to the designed interactive scenario.

Wizard of OZ The person who "plays computer" using paper mock-up or who "operates the puppet" using the mechanical mock-up can be a disturbing factor since the user may feel interacting with the person, not the robot. One way to overcome this is using the Wizard of OZ technique. Instead of operating the robot directly, the person plays as a “wizard” behind the scene and controls the robot remotely with a remote control, or from a connected computer. If controlled with a remote control, the “wizard” takes the role of the sensors by watching the user in a distance, and the role of the robot’s brain by make decisions for the robot. If controlled from a connected computer, the “wizard” only acts as the brain.

The prototype needed is more complicated than the mechanical mock-up. It has to be equipped with robotic hardware such as power supply, sensors, actuators and a minimal power of processing to control sensors and actuators. A connection to a remote computer is also needed.

To keep it easy and simple, these robotic components from the LEGO Mindstorms have been used in our projects. These components are shaped nicely to fit with other mechanical parts, so that the mechanical mock-ups can be upgraded to “Wizard of OZ” prototypes easily without building from the ground.

Prototypes with high fidelity of intelligence Using a human to take the role of the robot’s brain in above techniques pushes the vertical prototyping to an extreme – the robot tends to be smarter than it should be. In many cases there is a need to prototype the intelligent behavior with a higher fidelity, for example, to investigate how the appearance and the behavior match each other, to discover where the technical bottlenecks are, and to observe how the robot interacts with its physical environment. In short, it is possibly necessary to program the robot to enable its machine intelligence.

Many robots use a generic computer as its central processing unit. When it is too big to fit into the robots body, the computer is often “attached” to the robot via a wired or wireless connection. The advantage is that the programming environment is not limited by a specially designed robotic platform. The developer can choose whatever is convenient. The disadvantage is that the connection between the body and brain might become a bottleneck. The mobility of the robot is limited by the distance of the connection, and the performance is limited by the quality of the connection.

It would be better that the robot has its embodied processing unit that comes with an open and easy programming environment. This brings the LEGO Mindstroms on the table again. From the set, the RCX is programmable, microcontroller-based brick that can simultaneously operate three motors, three sensors, and an infrared serial communications interface. The enthusiasts have developed various kinds of firmware for it, which enables programming in Forth, C, and Java, turning the brick into an excellent platform for robotic prototyping (Bagnall, 2002).

Once the platform is selected, a good strategy of modeling and programming the robot helps to speed up the prototyping process. Considering the memory and processing power limitations of the RCX, the behavior-based AI model was used and a design pattern for the robots was developed in our projects. The behavior-based approach does not necessarily seek to produce cognition or a human-like thinking process. Instead of designing robots that could *think* intelligently, this approach aims at the robots that could *act* intelligently, with successful completion of a task as the goal. The similarty of the low level behavior of these robots leads us to develop an object-oriented design pattern that can be applied and reused.

A.3 Examples

Next four examples are introduced briefly, in which the rapid robotic prototyping has been successfully used to gain insight into user requirements. The first two examples are from this PhD project.

A.3.1 Tony

Tony is designed for an interactive television show as a companion robot for the audience in the NexTV project. Using LEGO Mindstorms, the first version of Tony (figure A.1(a)) was quickly built and shown to the users. The results from the user evaluation changed the role of Tony from a simple control device to a companion robot (figure A.1(b), figure A.1(c)). Tony watches the show together with the user, at the same time performing certain actions and behaviors according to the requests from the show and influencing the show back according the instructions from the user (Hu, 2001, 2003). More implementation details can also be found in chapter 11.

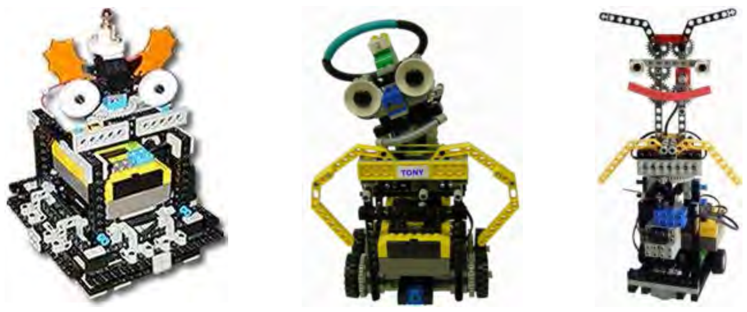

(a) version 1.0 (b) version 1.5 (c) version 2.0

Figure A.1: Tony

A.3.2 LegoMarine

LegoMarine (figure A.2 on the next page) is developed for the DeepSea application in the ICE-CREAM project. The 3D virtual world in the movie is extended and connected to the user's environment by distributing sound and lighting effects and using multiple displays and robotic interfaces. LegoMarine was designed and developed as the physical counterpart of a submarine in the virtual underwater. The user can direct the virtual submarine to navigate in the 3D space by tilting LegoMarine, or speed the submarine up by squeezing it. When the submarine hits something, LegoMarine also vibrates to give tactile feedback. The other behaviors of both are also synchronized, such as the speed of propeller and the intensity of the head lights.

At the beginning of the project, engineers from de Pinxi built an robotic submarine by putting sophisticated electronics into a shell striped from a toy submarine, but later found out it is impossible to frequently change the shape and functions according to the user's feedback. So the LegoMarine was born. LegoMarine does not move as many other robots does, but it is still considered as a robot because the device alone as a toy has its autonomous behavior, not only being reactive to user's interaction, but also being proactive to show certain behavior even when nobody is playing with it. It also adapts to the environment – if there is no other device around, it acts as a standalone toy, but if it is placed in a AmI environment it communicates with other devices to about the user interactions and listens to the requests from the others to perform and to behave. More implementation details of LegoMarine can be found in chapter 11.

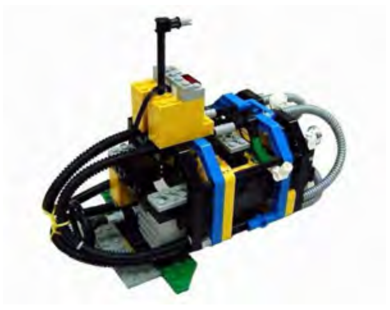


Figure A.2: LegoMarine

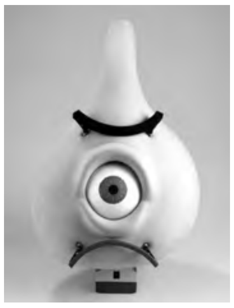


Figure A.3: eMuu

A.3.3 eMuu

eMuu (figure A.3) is intended to be an interface between an AmI home and its inhabitants (Bartneck, 2002). To gain acceptance in the homes of users the robot needs to be more than operational. The interaction needs to be enjoyable. The embodiment of the robot and its emotional expressiveness are key factors influencing the enjoyability of the interaction. Two embodiments (screen character and robotic character) were developed using the Muu robot (Okada, 2001) as the base for the implementation. The sophisticated technology of Muu was replaced with Lego Mindstorms equipment to quickly go though several prototyping cycles. The final version added an eyebrow and lip was added to the body to enable the robot to express emotions. The rapid robotic prototyping method was valuable to the project because the goal of the project was not to establish a convincing technical solution, rather, to experiment with different shapes and to find proper facial constructs for the evaluation how the embodied interaction and the robot's facial expressions would help the user to communicate with the AmI environments. The evaluation of the robot showed that the embodiment did not influence the enjoyability of the interaction, but the ability of a robot to express emotions had a positive influence.

A.3.4 Mr. Point, Mr. Ghost and Flow Breaker

These robots are developed for the research on the gaming experiences. Mr. Point (figure A.4(a)) and Mr. Ghost (figure A.4(b)) were created for the well-known Pac Man game, as support robots respectively for the player and the ghosts in the game (de Graaf and Feijs, 2002). From this study the researchers learned that it is difficult to attract the attention of the users to the perception space formed by the physical agents. This observation led the researcher to explore physical and on-screen strategies to break game flow in an effective and user acceptable way. The prototype of Mr. Point was reused again, but renamed to Flow Breaker (figure A.4(c)), with some modifications to fit onto the physical Pac Man maze (Eggen, Feijs, and Peters, 2003b). These studies were conducted by the researchers who are also specialists in electronics and software. To build prototypes quickly, it is not necessary for them to use robotic kits to speed up the process. Still, these robots reflect the principles of rapid robotic prototyping: reusing existing components, building simple prototypes quickly and evaluating with real users.

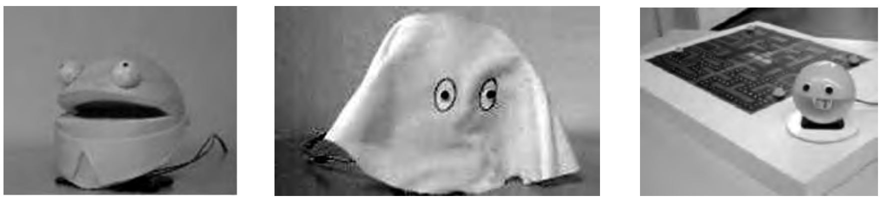

(a) Mr. Point (b) Mr. Ghost (c) Flow Breaker

Figure A.4: Game Support Robots

A.4 Discussion

Rapid prototyping is a powerful method for defining the user requirements for interactive robots, as it is in software engineering. Many prototyping techniques from software engineering are still valid, but need to be adjusted for the nature of robots and the tactile human-robot interaction. Non-specialists were encouraged to use robotic kits such as Lego Mindstorms to build robotic prototypes. In the experience of this project, using robotic kits simplifies and accelerates the prototyping process.

In this project, it also noticed that the Lego Mindstorms could not satisfy all the needs for rapid prototyping. Limited memory and speed of the processor, limited number of connected sensors and actuators, and poor infrared connections need a lot of improvements. It opens an opportunity for the industry to develop better robotic kits for prototyping and building interactive robots.

BACKGROUND MATERIAL B

StoryML DTD (Document Type Definition)

```
<?xml encoding="iso−8859−1"?>
<!−−
    XML document type definition (DTD) for StoryML 1.0.
    Date: 2003/05/30
    Author: Hu, Jun <j.hu@tue.nl>
−−>


<!−− Generally useful entities −−>
<!ENTITY % id−attr "id ID #IMPLIED">
<!ENTITY % title−attr " title CDATA #IMPLIED">
<!ENTITY % media−attr "
    src         CDATA #IMPLIED
    content     CDATA #IMPLIED
    actor IDREF #IMPLIED
    type (audio|video|audiovisual
          |text|image|behavior) 'audiovisual'
">

<!−− StoryML Document −−>
<!−−
  The root element StoryML contains all other elements.
−−>

<!ELEMENT StoryML (environment?,story?)>
<!ATTLIST StoryML
    %id−attr;
>

<!−− Environment Element −−>
<!−−
     Environment contains Presentation actors.
−−>
```

```
<!ELEMENT environment (actor*)>
<!ATTLIST environment
    %id-attr;
>

<!-- actor Element -->

<!ELEMENT actor EMPTY>
<!ATTLIST actor
    %id-attr;
    type (audiovisual|robot) "audiovisual"
>

<!-- Story Element -->

<!ELEMENT story (storyline *, interaction +)>
<!ATTLIST story
    %id-attr;
    %title-attr;
>

<!-- Storyline Element -->

<!ELEMENT storyline EMPTY>
<!ATTLIST storyline
    %id-attr;
    %media-attr;
>

<!-- Interaction Element -->

<!ELEMENT interaction (dialog*)>
<!ATTLIST interaction
    %id-attr;
>

<!-- Dialog Element -->

<!ELEMENT dialog (feedforward*, response*)>
<!ATTLIST dialog
    %id-attr;
    begin CDATA #IMPLIED
    end   CDATA #IMPLIED
    wait  CDATA #IMPLIED
    type  (immediate|delayed) "delayed"
>

<!-- Feedforward Element -->

<!ELEMENT feedforward EMPTY>
<!ATTLIST feedforward
    %id-attr;
    %media-attr;
>

<!-- Response Element -->
```

```
<!ELEMENT response (feedback*)>
<!ATTLIST response
          %id-attr;
     actor  IDREF #IMPLIED
     event         CDATA #IMPLIED
     switchto     IDREF #IMPLIED
     default      (yes|no)  "no"
>

<!-- Feedback Element -->

<!ELEMENT feedback EMPTY>
<!ATTLIST feedback
     %id-attr;
     %media-attr;
>
```

BACKGROUND MATERIAL C

TOONS in StoryML

```
<?xml version="1.0"?>
<! DOCTYPE StoryML SYSTEM "StoryML.dtd">
<!--
    This  is  the  XML document for TOONS.
    Date:  2003/05/30
    Author:  Hu,  Jun  <j.hu@tue.nl>
--->
<StoryML>
  <environment id="ToonsPlatform">
    <actor  id="tv"  type="audiovisual"  />
    <actor  id="Tony"    type="robot"  />
  </environment>

   <story  id="TOONS" title="TOONS (c) ID,  TU/e">
     < storyline  id="happygarden"
       src="rtp:// localhost/happygarden.mpg" ="tv"/>
     < storyline  id="angrygarden"
       src="rtp:// localhost/angrygarden.mpg" actor="tv"/>
     < interaction >
       <dialog  id="sleepTony" begin="03000ms" end="24000ms" >
         <feedforward content="sleeping"
           type="behavior" actor="Tony" />
       </ dialog>

       <dialog  id="wakeupTony" begin="24000ms" end="170000ms" >
         <feedforward content="awake"
           type="behavior" actor="Tony" />
       </ dialog>

       <dialog  id="whichdoortoenter"
         begin="34000ms" end="53000ms" wait="51000ms">
         <feedforward content="exciting"
           type="behavior" actor="Tony" />
         <response actor="Tony" event="left"
           switchto="happygarden" default="yes">
           <feedback id="openleftdoor"
             src="rtp:// localhost/left_door_open.mpg"
```

```
                type="video" actor="tv"  />
            </response>
            <response id="openrightdoor"
                actor="Tony" event="right"  switchto="angrygarden">
                <feedback id="openrightdoor"
                    src="rtp:// localhost/right_door_open.mpg"
                    type="video" actor="tv"  />
            </response>
        </dialog>

        <dialog  id="happyTony" begin="55000ms" end="170000ms"
            storyline ="happygarden">
            <feedforward content="happy"
                type="behavior" actor="Tony" />
        </dialog>

        <dialog  id="happyTonybeboring" begin="127000ms" end="170000ms"
            storyline ="happygarden">
            <feedforward content="sad"
                type="behavior" actor="Tony" />
        </dialog>

        <dialog  id="sadTony" begin="55000ms" end="120000ms"
            storyline ="angrygarden">
             <feedforward content="sad" type="behavior" actor="Tony" />
        </dialog>
      </ interaction >
    </ story>
</ StoryML>
```

BACKGROUND MATERIAL D

DTD of IPML Extension to SMIL

D.1 IPML layout module

```
<!−− IPML Layout Module = −−>
<!−− file: IPML−layout.mod

         This module extends the SMIL 2.0 Layout Module, by adding Actor
         elements.

         Date: 2004/10/30
         Author: Hu, Jun <j.hu@tue.nl>

         = −−>



<!−− = BasicLayout = −−>

<!−− = BasicLayout Profiling  Entities  = −−>

<!ENTITY % SMIL.layout. attrib          "">
<!ENTITY % SMIL.region. attrib          "">
<!ENTITY % SMIL.rootlayout. attrib      "">
<!ENTITY % IMIL.actor. attrib           "">

<!ENTITY % SMIL.layout.content       "EMPTY">
<!ENTITY % SMIL.region.content       "EMPTY">
<!ENTITY % SMIL.rootlayout.content  "EMPTY">
<!ENTITY % IPML.actor.content        "EMPTY">

<!−− = BasicLayout Entities  = −−>
<!ENTITY % SMIL.common−layout−attrs "
         height                  CDATA    'auto'
         width                   CDATA    'auto'
         %SMIL.backgroundColor.attrib;
">
```

```
<!ENTITY % SMIL.region—attrs "
          bottom                CDATA    'auto'
           left                 CDATA    'auto'
           right                CDATA    'auto'
          top                   CDATA    'auto'
          z—index               CDATA    #IMPLIED
          showBackground        (always|whenActive) 'always'
          %SMIL.fit. attrib ;
">

<!ENTITY % IPML.actor—attrs "
          type                  %URI.datatype;     #IMPLIED
">

<!—— = BasicLayout Elements = ——>

<!——
      Layout contains the region and root—layout elements defined by
      smil—basic—layout or other elements defined by an external layout
      mechanism.
——>

<!ENTITY % SMIL.layout.qname "layout">
<!ELEMENT %SMIL.layout.qname; %SMIL.layout.content;>
<!ATTLIST %SMIL.layout.qname; %SMIL.layout.attrib;
          %Core.attrib;
          %I18n.attrib;
          type CDATA 'text/smil—basic—layout'
>

<!—— = Region Element =——>

<!ENTITY % SMIL.region.qname "region">
<!ELEMENT %SMIL.region.qname; %SMIL.region.content;>
<!ATTLIST %SMIL.region.qname; %SMIL.region.attrib;
          %Core.attrib;
          %I18n.attrib;
          %SMIL.backgroundColor—deprecated.attrib;
          %SMIL.common—layout—attrs;
          %SMIL.region—attrs;
          regionName CDATA #IMPLIED
>


<!—— = Actor Element =——>

<!ENTITY % IPML.actor.qname "actor">
<!ELEMENT %IPML.actor.qname; %IPML.actor.content;>
<!ATTLIST %IPML.actor.qname; %IPML.actor.attrib;
          %Core.attrib;
          %I18n.attrib;
          %IPML.actor—attrs;
>


<!—— = Root—layout Element =——>

<!ENTITY % SMIL.root—layout.qname "root—layout">
```

```
<!ELEMENT %SMIL.root−layout.qname; %SMIL.rootlayout.content; >
<!ATTLIST %SMIL.root−layout.qname; %SMIL.rootlayout.attrib;
        %Core.attrib;
        %I18n.attrib;
        %SMIL.backgroundColor−deprecated.attrib;
        %SMIL.common−layout−attrs;
>


<!−− = AudioLayout = −−>

<!ENTITY % SMIL.AudioLayout.module "IGNORE">
<![%SMIL.AudioLayout.module;[
  <!−− = AudioLayout Entities = −−>
  <!ENTITY % SMIL.audio−attrs "
        soundLevel                    CDATA    '100&#37;'
  ">

  <!−− = AudioLayout Elements = −−>
  <!−− = Add soundLevel to region element = −−>
  <!ATTLIST %SMIL.region.qname; %SMIL.audio−attrs;>
 ]]> <!−− end AudioLayout.module −−>


<!−− = MultiWindowLayout = −−>

<!ENTITY % SMIL.MultiWindowLayout.module "IGNORE">
<![%SMIL.MultiWindowLayout.module;[
  <!−− = MultiWindowLayout Profiling Entities = −−>
  <!ENTITY % SMIL.topLayout.attrib      "">
  <!ENTITY % SMIL.topLayout.content    "EMPTY">

  <!−− = MultiWindowLayout Elements = −−>
  <!−−= topLayout element = −−>
  <!ENTITY % SMIL.topLayout.qname "topLayout">
  <!ELEMENT %SMIL.topLayout.qname; %SMIL.topLayout.content;>
  <!ATTLIST %SMIL.topLayout.qname; %SMIL.topLayout.attrib;
        %Core.attrib;
        %I18n.attrib;
        %SMIL.common−layout−attrs;
        close              (onRequest|whenNotActive) 'onRequest'
        open               (onStart|whenActive)      'onStart'
  >
 ]]> <!−− end MultiWindowLayout.module −−>


<!−− = HierarchicalLayout = −−>

<!ENTITY % SMIL.HierarchicalLayout.module "IGNORE">
<![%SMIL.HierarchicalLayout.module;[
  <!−− = HierarchicalLayout Profiling Entities = −−>
  <!ENTITY % SMIL.regPoint.attrib       "">
  <!ENTITY % SMIL.regPoint.content     "EMPTY">

  <!−− = HierarchicalLayout Elements = −−>
  <!ENTITY % SMIL.regPoint.qname "regPoint">
  <!ELEMENT %SMIL.regPoint.qname; %SMIL.regPoint.content;>
  <!ATTLIST %SMIL.regPoint.qname; %SMIL.regPoint.attrib;
```

```
          %Core.attrib;
          %I18n.attrib;
          %SMIL.regAlign.attrib;
          bottom                CDATA    'auto'
           left                 CDATA    'auto'
          right                 CDATA    'auto'
          top                   CDATA    'auto'
  >
]]>  <!-- end HierarchicalLayout.module -->


<!-- end of IPML-layout.mod -->
```

D.2 IPML Media Object Module

```
<!-- IPML Media Objects Modules = -->
<!-- file:  IPML-media.mod

          This  module extends the SMIL 2.0 Media Object Module, by adding Action
          elements.

          Date:  2004/10/30
          Author:  Hu, Jun <j.hu@tue.nl>

      = -->



<!-- = Profiling  Entities  = -->

<!ENTITY % SMIL.MediaClipping.module "IGNORE">

<![%SMIL.MediaClipping.module;[
  <!ENTITY % SMIL.mo-attributes-MediaClipping "
          %SMIL.MediaClip.attrib;
  ">
]]>

<!ENTITY % SMIL.mo-attributes-MediaClipping "">

<!ENTITY % SMIL.MediaClipping.deprecated.module "IGNORE">

<![%SMIL.MediaClipping.module;[
  <!ENTITY % SMIL.mo-attributes-MediaClipping-deprecated "
          %SMIL.MediaClip.attrib.deprecated;
  ">
   ]]>

<!ENTITY % SMIL.mo-attributes-MediaClipping-deprecated "">

<!ENTITY % SMIL.MediaParam.module "IGNORE">
<![%SMIL.MediaParam.module;[
  <!ENTITY % SMIL.mo-attributes-MediaParam "
          erase          (whenDone|never)         'whenDone'
          mediaRepeat (preserve| strip )          'preserve'
           sensitivity      CDATA              'opaque'
  ">
  <!ENTITY % SMIL.param.qname "param">
  <!ELEMENT %SMIL.param.qname; EMPTY>

  <!ATTLIST %SMIL.param.qname; %SMIL.param.attrib;
    %Core.attrib;
    %I18n.attrib;
    name         CDATA            #IMPLIED
    value        CDATA            #IMPLIED
    valuetype    (data|ref|object)  "data"
    type         %ContentType.datatype;  #IMPLIED
  >
 ]]>

<!ENTITY % SMIL.mo-attributes-MediaParam "">

<!ENTITY % SMIL.MediaAccessibility.module "IGNORE">

<![%SMIL.MediaAccessibility.module;[
  <!ENTITY % SMIL.mo-attributes-MediaAccessibility "
        readIndex     CDATA             #IMPLIED
  ">
 ]]>

<!ENTITY % SMIL.mo-attributes-MediaAccessibility "">

<!ENTITY % SMIL.BasicMedia.module "INCLUDE">

<![%SMIL.BasicMedia.module;[
  <!ENTITY % SMIL.media-object.content "EMPTY">
  <!ENTITY % SMIL.media-object.attrib "">

  <!-- = Media Objects Entities = -->

  <!ENTITY % SMIL.mo-attributes-BasicMedia "
        src                CDATA   #IMPLIED
        type               CDATA   #IMPLIED
  ">

  <!ENTITY % SMIL.mo-attributes "
        %Core.attrib;
        %I18n.attrib;
        %SMIL.Description.attrib;
        %SMIL.mo-attributes-BasicMedia;
        %SMIL.mo-attributes-MediaParam;
        %SMIL.mo-attributes-MediaAccessibility;
        %SMIL.media-object.attrib;
  ">

  <!ENTITY % IPML.action.content  "(%IPML.event.qname;)*">

  <!ENTITY % IPML.action-attributes "
        %Core.attrib;
        %I18n.attrib;
        %SMIL.Description.attrib;
        %SMIL.media-object.attrib;
        src                CDATA                #IMPLIED
        type               %URI.datatype;       #IMPLIED
```

```
      observe           CDATA                #IMPLIED
">

<!--
    Most info  is  in  the  attributes ,  media objects are  empty or
    have children  defined at  the language integration   level:
-->

<!ENTITY % SMIL.mo-content "%SMIL.media-object.content;">

<!-- = Media Objects Elements = -->
<!ENTITY % SMIL.ref.qname          "ref">
<!ENTITY % SMIL.audio.qname        "audio">
<!ENTITY % SMIL.img.qname          "img">
<!ENTITY % SMIL.video.qname        "video">
<!ENTITY % SMIL.text.qname         "text">
<!ENTITY % SMIL.textstream.qname "textstream">
<!ENTITY % SMIL.animation.qname "animation">
<!ENTITY % IPML.action.qname       "action">


<!ENTITY % SMIL.ref.content          "%SMIL.mo-content;">
<!ENTITY % SMIL.audio.content        "%SMIL.mo-content;">
<!ENTITY % SMIL.img.content          "%SMIL.mo-content;">
<!ENTITY % SMIL.video.content        "%SMIL.mo-content;">
<!ENTITY % SMIL.text.content         "%SMIL.mo-content;">
<!ENTITY % SMIL.textstream.content "%SMIL.mo-content;">
<!ENTITY % SMIL.animation.content  "%SMIL.mo-content;">
<!ENTITY % IPML.action.content       "%IPML.action.content;">


<!ELEMENT %SMIL.ref.qname;          %SMIL.ref.content;>
<!ELEMENT %SMIL.audio.qname;        %SMIL.audio.content;>
<!ELEMENT %SMIL.img.qname;          %SMIL.img.content;>
<!ELEMENT %SMIL.video.qname;        %SMIL.video.content;>
<!ELEMENT %SMIL.text.qname;         %SMIL.text.content;>
<!ELEMENT %SMIL.textstream.qname;  %SMIL.textstream.content;>
<!ELEMENT %SMIL.animation.qname;   %SMIL.animation.content;>
<!ELEMENT %IPML.action.qname;       %IPML.action.content;>


<!ATTLIST %SMIL.img.qname;
          %SMIL.mo-attributes;
>

<!ATTLIST %SMIL.text.qname;
          %SMIL.mo-attributes;
>

<!ATTLIST %SMIL.ref.qname;
          %SMIL.mo-attributes-MediaClipping;
          %SMIL.mo-attributes-MediaClipping-deprecated;
          %SMIL.mo-attributes;
>

<!ATTLIST %SMIL.audio.qname;
          %SMIL.mo-attributes-MediaClipping;
          %SMIL.mo-attributes-MediaClipping-deprecated;
          %SMIL.mo-attributes;
  >

  <!ATTLIST %SMIL.video.qname;
          %SMIL.mo-attributes-MediaClipping;
          %SMIL.mo-attributes-MediaClipping-deprecated;
          %SMIL.mo-attributes;
  >

  <!ATTLIST %SMIL.textstream.qname;
          %SMIL.mo-attributes-MediaClipping;
          %SMIL.mo-attributes-MediaClipping-deprecated;
          %SMIL.mo-attributes;
  >

  <!ATTLIST %SMIL.animation.qname;
          %SMIL.mo-attributes-MediaClipping;
          %SMIL.mo-attributes-MediaClipping-deprecated;
          %SMIL.mo-attributes;
  >

  <!ATTLIST %IPML.action.qname;
          %IPML.action-attributes%;
  >

]]>
<!ENTITY % SMIL.mo-attributes-BasicMedia "">

<!-- BrushMedia -->
<!ENTITY % SMIL.BrushMedia.module "IGNORE">
<![%SMIL.BrushMedia.module;[
  <!ENTITY % SMIL.brush.attrib  "">
  <!ENTITY % SMIL.brush.content "%SMIL.mo-content;">
  <!ENTITY % SMIL.brush.qname "brush">
  <!ELEMENT %SMIL.brush.qname; %SMIL.brush.content;>
  <!ATTLIST %SMIL.brush.qname; %SMIL.brush.attrib;
        %Core.attrib;
        %I18n.attrib;
        %SMIL.Description.attrib;
        %SMIL.mo-attributes-MediaAccessibility;
        %SMIL.mo-attributes-MediaParam;
        %SMIL.media-object.attrib;
        color          CDATA            #IMPLIED
  >
]]>

<!-- end of IPML-media.mod -->
```

D.3 IPML Linking Module

```
<!-- IPML Linking Module = -->
<!-- file: IPML-link.mod

        This module extends the SMIL 2.0 Linking Module, by adding Event
        elements.

        Date: 2004/10/30
        Author: Hu, Jun <j.hu@tue.nl>

     = -->



<!-- = LinkingAttributes Entities = -->
<!ENTITY % SMIL.linking-attrs "
        sourceLevel           CDATA                 '100&#37;'
        destinationLevel      CDATA                 '100&#37;'
        sourcePlaystate       (play|pause|stop)     #IMPLIED
        destinationPlaystate  (play|pause)          'play'
        show                  (new|pause|replace)   'replace'
        accesskey             %Character.datatype;  #IMPLIED
        target                CDATA                 #IMPLIED
        external              (true|false)          'false'
        actuate               (onRequest|onLoad)    'onRequest'
        %SMIL.tabindex.attrib;
">


<!-- = BasicLinking Elements = -->
<!ENTITY % SMIL.BasicLinking.module "IGNORE">
<![%SMIL.BasicLinking.module;[

  <!-- = BasicLinking Entities = -->
  <!ENTITY % SMIL.Shape "(rect|circle|poly|default)">
  <!ENTITY % SMIL.Coords "CDATA">
    <!-- comma separated list of lengths -->

  <!ENTITY % SMIL.a.attrib  "">
  <!ENTITY % SMIL.a.content "EMPTY">
  <!ENTITY % SMIL.a.qname  "a">
  <!ELEMENT %SMIL.a.qname; %SMIL.a.content;>
  <!ATTLIST %SMIL.a.qname; %SMIL.a.attrib;
    %SMIL.linking-attrs;
    href                     %URI.datatype;     #REQUIRED
    %Core.attrib;
    %I18n.attrib;
  >

  <!ENTITY % SMIL.area.attrib  "">
  <!ENTITY % SMIL.area.content "EMPTY">
  <!ENTITY % SMIL.area.qname  "area">
  <!ELEMENT %SMIL.area.qname; %SMIL.area.content;>
  <!ATTLIST %SMIL.area.qname; %SMIL.area.attrib;
    %SMIL.linking-attrs;
    shape                    %SMIL.Shape;          'rect'
    coords                   %SMIL.Coords;         #IMPLIED
    href                     %URI.datatype;     #IMPLIED
    nohref                   (nohref)           #IMPLIED
    %Core.attrib;
    %I18n.attrib;
  >


  <!ENTITY % IPML.event.attrib  "">
  <!ENTITY % IPML.event.content "EMPTY">
  <!ENTITY % IPML.event.qname  "event">
  <!ELEMENT %IPML.event.qname; %IPML.event.content;>
  <!ATTLIST %IPML.event.qname; %IPML.event.attrib;
    %SMIL.linking-attrs;
    href                          %URI.datatype;       #IMPLIED
    nohref                        (nohref)             #IMPLIED
    %Core.attrib;
    %I18n.attrib;
  >

  <!ENTITY % SMIL.anchor.attrib   "">
  <!ENTITY % SMIL.anchor.content "EMPTY">
  <!ENTITY % SMIL.anchor.qname "anchor">
  <!ELEMENT %SMIL.anchor.qname; %SMIL.anchor.content;>
  <!ATTLIST %SMIL.anchor.qname; %SMIL.anchor.attrib;
    %SMIL.linking-attrs;
    shape                         %SMIL.Shape;         'rect'
    coords                        %SMIL.Coords;        #IMPLIED
    href                          %URI.datatype;       #IMPLIED
    nohref                        (nohref)             #IMPLIED
    %Core.attrib;
    %I18n.attrib;
  >
]]>  <!-- end of BasicLinking -->

<!-- = ObjectLinking = -->
<!ENTITY % SMIL.ObjectLinking.module "IGNORE">
<![%SMIL.ObjectLinking.module;[

  <!ENTITY % SMIL.Fragment "
    fragment                      CDATA                #IMPLIED
  ">

  <!-- = ObjectLinking Elements = -->
  <!-- add fragment attribute to  area,  and anchor elements -->
  <!ATTLIST %SMIL.area.qname;
      %SMIL.Fragment;
  >

  <!ATTLIST %SMIL.anchor.qname;
      %SMIL.Fragment;
  >
]]>
<!-- = End ObjectLinking = -->

<!-- end of IPML-link.mod -->
```

D.4 IPML Language Profile DTD driver

```
<!-- IPML DTD -->
<!-- file:  IPML.dtd -->
<!-- IPML DTD

            IPML is  an extension  of  SMIL 2.0,  by  replacing  its  layout
            module, Media Object module, and Linking module.

            Date:  2004/10/30
            Author: Hu, Jun <j.hu@tue.nl>

-->



<!ENTITY % NS.prefixed "IGNORE" >
<!ENTITY % SMIL.prefix  ""  >

<!-- Define the Content Model -->
<!ENTITY % smil-model.mod
    PUBLIC "-//W3C//ENTITIES SMIL 2.0 Document Model 1.0//EN"
           "smil-model-1.mod" >

<!-- Modular Framework Module     ..................................     -->
<!ENTITY % smil-framework.module "INCLUDE" >
<![%smil-framework.module;[
<!ENTITY % smil-framework.mod
     PUBLIC "-//W3C//ENTITIES SMIL 2.0 Modular Framework 1.0//EN"
            "smil-framework-1.mod" >
%smil-framework.mod;]]>

<!-- The IPML includes the following  sections:
             C. The SMIL Animation Module
             D. The SMIL Content Control  Module
             G. The IPML Layout Module
             H. The IPML Linking  Module
             I. The IPML Media Object Module
             J. The SMIL Metainformation Module
             K. The SMIL Structure  Module
             L. The SMIL Timing and Synchronization Module
             M. Integrating  SMIL Timing into  other  XML-Based Languages
             P. The SMIL Transition   effects  Module

             The SMIL Streaming Media Object Module is optional.
-->

<!--
<!ENTITY % smil-streamingmedia.model "IGNORE">
<![%smil-streamingmedia.model;[
  <!ENTITY % smil-streaming-mod
    PUBLIC "-//W3C//ELEMENTS SMIL 2.0 Streaming Media Objects//EN"
    "SMIL-streamingmedia.mod">
  %smil-streaming-mod;
 ]]>
-->

<!ENTITY % SMIL.anim-mod
  PUBLIC "-//W3C//ELEMENTS SMIL 2.0 Animation//EN"
  "SMIL-anim.mod">
<!ENTITY % SMIL.control-mod
  PUBLIC "-//W3C//ELEMENTS SMIL 2.0 Content Control//EN"
  "SMIL-control.mod">
<!ENTITY % IPML.layout-mod
  SYSTEM "IPML-layout.mod">
<!ENTITY % IPML.link-mod
  SYSTEM "IPML-link.mod">
<!ENTITY % IPML.media-mod
```

```
  SYSTEM "SMIL−media.mod">
<!ENTITY % SMIL.meta−mod
  PUBLIC "−//W3C//ELEMENTS SMIL 2.0 Document Metainformation//EN"
  "SMIL−metainformation.mod">
<!ENTITY % SMIL.struct−mod
  PUBLIC "−//W3C//ELEMENTS SMIL 2.0 Document Structure//EN"
  "SMIL−struct.mod">
<!ENTITY % SMIL.timing−mod
  PUBLIC "−//W3C//ELEMENTS SMIL 2.0 Timing//EN"
  "SMIL−timing.mod">
<!ENTITY % SMIL.transition−mod
  PUBLIC "−//W3C//ELEMENTS SMIL 2.0 Transition//EN"
  "SMIL−transition.mod">

%SMIL.struct−mod;
%SMIL.anim−mod;
%SMIL.control−mod;
%SMIL.meta−mod;
%IPML.layout−mod;
%IPML.link−mod;
%IPML.media−mod;
%SMIL.timing−mod;
%SMIL.transition−mod;

<!−− end of IPML.dtd −−>
```

BACKGROUND MATERIAL E

Specifications of the Timed Action Pattern

This background material presents the formal specification of the components of the Timed Action pattern that is introduced in section 6.1 of chapter 6.

E.1 *ActionService*

An *ActionService* defines the behavior and the state of an action service supplier and implements the actual actions on its state element – in the IPML system it is often a media object. Let's first introduce a type that identifies each action:

$$ActionID == 1 \mathrel{..} n$$

with which $n : \mathbb{N}$ operations are assumed to be served as the interface to the actions, and each operation has a name Op_i that can be identified by an $i \in ActionID$. The *ActionService* can then be modeled in Object-Z as follows:

ActionService

$$\begin{array}{|l}
Op_1 \mathrel{\widehat=} [p?, r! : Dictionary] \\
Op_2 \mathrel{\widehat=} [p?, r! : Dictionary] \\
\ldots \\
Op_n \mathrel{\widehat=} [p?, r! : Dictionary] \\
\hline
\end{array}$$

Each Op_i ($i \in ActionID$) operation accepts a set of parameters $p?$ and produces a set of $r!$. Since it is not yet known what exactly the input parameters and the output results are, both parameters and results are modeled as of type Dictionary:

$$Dictionary == String \mathrel{+\!\!\!\!\to} \mathbb{O}$$

that contains *name* $\mapsto$ *value* pairs, where *name* is of type

$$[String]$$

that declares all possible text strings, and *value* is an object of any type. The symbol $\mathbb{O}$ is defined as

$$\mathbb{O} == \downarrow Object$$

where *Object* is the root superclass for all the classes in the system. $\downarrow Object$ is the set of all the objects of the class *Object* and its subclasses – the object universe of the system. Hence $\mathbb{O}$ denotes the object universe (Duke and Rose, 2000; Smith, 2000).

Passing named parameters and returning named values are a common technique in object oriented systems to invoke functions and get the results so that the parameters and the results do not have to be explicitly defined in advance, nor in a fixed order. The invoked function gets the input values by looking up the parameters dictionary using the name string as the keyword. The invoking procedure gets the output values from the function in the same way. Some object oriented languages, for example Python(Pilgrim, 2004), natively integrate this technique as an option for function definition and invocation.

The technique used here supports passing and returning values of any object types, but not primary types. To extend this technique for all types of values, a global total function $\mathcal{O}$ is defined as

$$\begin{array}{l}[X]\\ \hline \mathcal{O} : X \rightarrow \mathbb{O} \\ \hline \forall\, x : X \bullet \textbf{if}\; x \in \mathbb{O} \;\textbf{then}\; \mathcal{O}(x) = x \\ \qquad\qquad \textbf{else}\; \mathcal{O}(x) \in Wrapper[X] \land \mathcal{O}(x).x = x \end{array}$$

that turns any type of value to an object, where *Wrapper* is a simple class that wraps a single state variable *x* of type *X*:

$$\begin{array}{l} Wrapper[X] \\ \hline \boxed{x : X} \end{array}$$

Let's also define another global total function $\mathcal{V}$:

$$\begin{array}{l}[X]\\ \hline \mathcal{V} : \mathbb{O} \rightarrow X \\ \hline \forall\, o : \mathbb{O} \bullet \textbf{if}\; o \in Wrapper[X] \;\textbf{then}\; \mathcal{V}(o) = o.x \\ \qquad\qquad \textbf{else}\; \mathcal{V}(o) = o \end{array}$$

The function $\mathcal{O}$ works as an encoder to transform values of any type to an object, and the function $\mathcal{V}$ works as a decoder to retrieve the original value.

For convenience, two general operations are also defined to put a $name \mapsto value$ pair into a *Dictionary* and to consult a *Dictionary* to get the value according to the given *name*:

$$
\begin{array}{l}
[X] \\
\hline
Put : Dictionary \times String \times X \rightarrow Dictionary \\
Get : Dictionary \times String \rightarrow X_{\circledcirc} \\
\hline
\forall\, d : Dictionary;\ s : String;\ v : X \bullet \\
\qquad Put = d \oplus \{s \mapsto \mathcal{O}(v)\} \\
\forall\, d : Dictionary;\ s : String \bullet \\
\qquad Get = \mathbf{if}\ s \in \mathrm{dom}(d)\ \mathbf{then}\ \mathcal{V}(d(s))\ \mathbf{else}\ \mathsf{null}
\end{array}
$$

$Put(d, s, v)$ returns a new dictionary which agrees the old Dictionary d everywhere except the entry of s; but also adds the new entry $s \mapsto \mathcal{O}(v)$. $Get(d, s)$ returns the value of the entry s if the Dictionary d contains such an entry, otherwise returns null. The operator $\oplus$ applies the relational overriding of two relations of the same type. For example, if Q and R are relations of the same type, $Q \oplus R$ is a relation that agrees with Q everywhere outside of the domain of R, but agree with R where R is defined (ISO/IEC, 2002).

E.2 *Action*

To execute a particular operation from *ActionService* asynchronously, the operation need to be stored, identified, and retrieved together with its input parameters. An often used technique in this situation, described in the GoF book (Gamma et al., 1995) as the Command pattern, encapsulates the operations in objects, to control their selection and sequencing, and otherwise manipulate them. This technique is used to wrap the operations from *ActionService* as objects. First an *Action* class is introduced:

$$
\begin{array}{ll}
\multicolumn{2}{l}{Action} \\
\hline
eq : ExecutionQueue & \\
as : \downarrow ActionService & \\
p : Dictionary & [\,\text{Parameters}] \\
tr : TentativeResult & [\,\text{Tentative result of this action}] \\
\hline
\multicolumn{2}{l}{\textit{INIT}} \\
\multicolumn{2}{l}{tr.\textit{INIT}} \\
\hline
\multicolumn{2}{l}{Execute \mathrel{\widehat{=}} [\ ]} \\
\multicolumn{2}{l}{Enqueue \mathrel{\widehat{=}} eq.Enqueue(self)}
\end{array}
$$

and an abbreviation

$$Action(I) ==$$

$Action_I$

$Action$

$Execute \mathrel{\widehat=} as.Op_I(p) \mathbin{⨟} tr.Complete$

where I is used as a generic parameter for the abbreviation. For each Op_i ($i \in ActionID$) in *ActionService*, a corresponding class $Action(i)$ is then defined, which extends the class *Action* and implements the operation *Execute* to invoke corresponding operation $as.Op_i$.

Every *Action* object has a backward reference "*as*" to the associated ActionService object, a $p : Dictionary$ to store the input parameters, and a $tr : TentativeResult$ (see *TentativeResult* on page 275) to notify the others the completion of the operation, or for others to query the operation output. The initial state of an *Action* object requires the attribute $tr : TentativeResult$ to be initialized first, which is to set the state $tr.tentative$ to be true. The operation *Execute* outputs the stored parameter list as input for the corresponding operation Op_I in "*as*". The result of the operation is in turn used as input for *tr* to complete.

An $Action(i)$ ($i \in ActionID$) object can be executed directly by invoking the operation *Execute*, however there is a problem: once an $Action(I)$ object is in execution, other $Action(i)$ ($i \in ActionID$) objects will be blocked from execution if they are invoked from the same process. This can cause a performance bottleneck in scheduling unless the action execution is asynchronously separated from the scheduling process. The solution is to employ an execution queue with a process that is independent of the scheduling process.

Every *Action* object has a reference $eq : ExecutionQueue$ to such a process. Instead of the operation *Execute*, the operation *Enqueue* puts the object itself into the execution queue and return immediately, leaving the execution task to the process of *eq*.

E.3 *ExecutionQueue*

An *ExecutionQueue* object has an independent process, accepting *Action* objects into its queue and dequeuing them for execution:

$ExecutionQueue$

$\upharpoonright(\mathrm{I{\scriptsize NIT}}, Enqueue)$

$todo : \mathrm{seq} \downarrow Action$

$\mathrm{I{\scriptsize NIT}}$: $todo = \langle\,\rangle$

$Enqueue$

$\Delta(todo)$

$a? : \downarrow Action$

$todo' = todo \mathbin{^\frown} \langle a? \rangle$

$Dequeue$

$\Delta(todo)$

$a! : \downarrow Action$

$todo = \langle a! \rangle \mathbin{^\frown} todo'$

IdleTick

$\neg\ \mathrm{pre}(\mathit{Execute})$

$\tau' = \tau + 1$

$\mathit{Execute} \mathrel{\widehat=} (\mathit{Dequeue} \ \bullet\ a!.\mathit{Execute}) \setminus (a!)$

$\mathrm{P{\scriptsize ROCESS}} \mathrel{\widehat=} \mu\, \mathit{EQ} \bullet (\mathit{Execute} \mathbin{[]} \mathit{IdleTick}) \mathbin{⨾} \mathit{EQ}$

The only visible Operation of the class *ActionService*, specified with a visibility list $\upharpoonright(\mathrm{I{\scriptsize NIT}}, \mathit{Enqueue})$, is the operation *Enqueue* that takes an object $a? : \downarrow\!\mathit{Action}$ as input and puts it in a first-in-first-out (FIFO) queue *todo*. Notice that $\mathrm{I{\scriptsize NIT}}$ is the initial state, not an operation. The operation *Dequeue* dequeues and outputs an *Action* object from the queue if the queue is not empty. The predicate $\mathit{todo} = \langle a! \rangle \frown \mathit{todo}'$ requires *todo* to have at least one element, otherwise $\langle\,\rangle = \langle a! \rangle \frown \mathit{todo}'$ would always result in false.

The operation *Execute* dequeues an object $a! : \downarrow\!\mathit{Action}$ and invokes the *Execute* operation in $a!$. The scope enrichment operator $\bullet$ is used to enrich the environment of the rest of the operation *Execute* with the auxiliary output variable $a!$ from *Dequeue*. The hiding operator $\setminus$ at the end hides the output $a!$ from the scope enrichment operation. So the operation *Execute* as a whole does not communicate with its environment.

The class *ActionService* has to have an active process to dequeue the actions and execute them automatically. Considering that later a notion of time also needs to be incorporated, the following gives the Object-Z extension that models real-time active processes in a system, introduced by Dong, Colton, and Zucconi (1996), in which the basic time type is modeled as natural numbers

$$\mathbb{T} == \mathbb{N}$$

that represents the absolute discrete time domain. Let's assume every Object-Z class implicitly has a common superclass *Object* as the root of the class hierarchy:

Object

Δ

$\tau : \mathbb{T}$

Every Object-Z class then inherits the secondary attribute $\tau : \mathbb{T}$ (originally called $\mathit{now} : \mathbb{T}$ by Dong et al. (1996)) to represent the absolute current time such that the absolute time is shared by every object in a system:

$$\forall\, o_1, o_2 : \mathbb{O} \bullet o_1.\tau = o_2.\tau$$

As a secondary attribute, τ is included in the Δ-list of the state schema of every object - it may be changed by any operation. Every operation of any Object-Z class also implicitly includes the predicate

$$\tau' \geqslant \tau$$

to ensure time cannot go backwards.

With the timing extension to Object-Z, the class *ActiveService* can then be modeled. It has a process of its own, which runs in a different thread than its client's. The operation PROCESS is a recursive non-terminating process in a finite representation form. It describes a unique continuous sequence of the operations, including the operation *IdleTick* that does nothing but takes one time unit when no other operation meets its precondition. The predicate $\neg\ \mathrm{pre}(Execute)$ ensures that the operation *IdleTick* can occur only when the operation *Execute* can not occur.

Dong et al. (1996) used this technique only in the top level system class to model the real time behavior. It is extended to model all the active objects in a system, in which the appearance of the continuous operation PROCESS identifies an active object as the operation MAIN does in Timed Communication Object-Z (TCOZ) (Mahony and Dong, 2000). Since it is modeled as an indefinite recursive loop, the post state of this operation can never be reached, neither can the post conditions be evaluated. Hence it makes no sense to use this operation in other classes and operations. As a consequence, the PROCESS operation does not communicate with its environment – it has no input nor output. It is to be scheduled and run by the system. The PROCESS operations of all the active objects in a system are assumed to run in parallel. Since the appearance of PROCESS makes an object already "active", the operation PROCESS implicitly includes the state initialization INIT. As in TCOZ, inheriting an active object does not automatically makes an subtype active, unless PROCESS is explicitly included in the definition of the subtype.

In *ExecutionQueue*, the PROCESS operation

$$\mu\, EQ \bullet (Execute \mathbin{[]} IdleTick) \mathbin{⨟} EQ$$

means there is a unique operation *EQ* that first executes the operation *Execute* if its preconditions are met, or the operation *IdleTick* for one time unit otherwise, then executes *EQ* again. The choice operator $[]$ defines an operation that makes an angelic choice between two given operations.

The sequential composition operator $⨟$ in the operation PROCESS guarantees the synchronized access to the operation *Execute* and hence the action operations – none of them will be executed in parallel. In the case of serving the actions on a media object through an *ActionService* object, actions are easily synchronized in parallel with the media object thread in which the element is busy with other processes. This will free us from the worry whether the media object has internal synchronization mechanisms.

E.4 *TimedAction*

For scheduling an *Action* object to be executed at a planned time, the object must be associated with that time. The association can be made inside the scheduler by maintaining a list of maplets and updating the maplets as the system time changes. The change update mechanism in the Observer pattern may be reduced, but then the maplets have to be wrapped up as an *Observer* object:

TimedAction

$Observer[Update_o / Update]$

$t : \mathbb{T}$ [Planned execution time]
$a : \downarrow Action$ [Action to do]

$Update \mathrel{\widehat{=}} Update_o \mathbin{⨟} a.Enqueue$

A *TimedAction* wraps an *Action* with an attribute of planned execution time. However having the time does not mean it knows the time. It can of course be implemented as an active object that has a PROCESS polling the current system time and execute itself if the planned time has reached. But as the number of the *TimedAction* increases, too many active processes may become a bottleneck of the system performance. Instead of implementing each *TimedAction* as an active object, the system rather centralize the scheduling in a *Scheduler* for all the *TimedAction* objects. Every *TimedAction* object needs to be registered to the *Scheduler* and the *Scheduler* notifies the observing *TimedAction* objects when their planned execution times have been reached. The Observer pattern is applied to the *Scheduler* and the *TimeAction* components to realize this timing dependency. The *TimedAction* overrides the operation *Update* in *Observer* (see *Observer* on page 62): it invokes the operation $Update_0$ (renamed from the operation *Update* in its super class *Observer*) to consume the input $o? : Observable$, followed by the operation $a.Enqueue$ to add the *Action* component a to the *todo* list in *ExecutionQueue*.

Instead of including an *Action* object as an attribute, the class *TimeAction* might also inherit the class *Action* to implement the relation between the action and the planned time. However doing so would put the class *TimedAction* at the same level of $Action(i)$ $(i \in ActionID)$ in the inheritance tree, which would cause difficulties to identify which Op_i $(i \in ActionID)$ a *TimedAction* is related to.

E.5 *Scheduler*

A *Scheduler* is an *Observable* (see definition on page 62) object:

Scheduler

$\upharpoonright(\text{INIT}, Subscribe, Unsubscribe)$
$Observable[Notify_{obl} / Notify]$
$tas == obs \cap TimedAction$ [TimedAction observers]

IdleTick
$\neg\ \mathrm{pre}(Notify)$
$\tau' = \tau + 1$

$$Notify \mathrel{\widehat{=}} \bigwedge ta : tas \mid ta.t = min(\{x : tas \bullet x.t\}) \land ta.t \leqslant \tau \bullet ta.Update(self) \ \land\ Unsubscribe(ta)$$

$$\text{PROCESS} \mathrel{\widehat{=}} \mu\, S \bullet (Notify \mathbin{[]} IdleTick) \mathbin{⨟} S$$

The class *Scheduler* extends the class *Observable* (see *Observable* on page 62), in which the operation *Notify* is canceled and redefined to decide which *TimedAction* objects to execute and unsubscribe next, instead of notifying all the managed observers. Only *TimedAction* observers ($tas == obs \cap TimedAction$) whose planned execution times are the earliest ($ta.t = min(\{x : tas \bullet x.t\})$ in the list and have been passed ($ta.t \leqslant \tau$) will be notified and then unsubscribed from the waiting list. The relation between the observable (*Scheduler* objects) and the observer (*TimedAction* objects) are tighter than what is offered by the *Observer* pattern – The *Scheduler* knows it is managing the *TimedAction* objects, not general *Observer* objects.

A design decision has been made here to separate the action scheduling process and the action execution process. The *TimedAction*s might have been designed to be executed in the process of the scheduler instead of leaving it to the *ActionService* to reduce the number of active processes. To ensure synchronized access to the actions, the update notification to the observers has then to be sequential, instead of the simultaneous composition $\bigwedge$ in the operation *Notify*. However, if many actions requires to be executed at the same time, the performance of the scheduler is crucial and may become a bottleneck of the overall parallelism.

Another benefit of this separation is the possibility of sharing a scheduler among multiple action services. What the scheduler manages are *TimedAction* objects. It does not require these objects to be from the same *ActionService* object. Sharing the scheduler can decrease the number of active process. However, it may also increase the workload of the scheduler. Whether to share a scheduler among action services depends on on the capability of the physical system and the requirements on the actual performance. Here let's leave the decision to implementation.

E.6 *ActionServiceProxy*

The class *ActionServiceProxy* applies the Proxy pattern (Gamma et al., 1995) to hide complexity of scheduling and access control, and to provide an interface that allows clients to invoke the public operations to plan the actions as if these actions are accessible directly from the *ActionService*.

It is necessary to model the creation of the *TimedAction* objects, but Object-Z does not address the creation or the destruction of objects per se: it is asserted that objects always exist (Duke and Rose, 2000, p.21). However, the creation of an object can be modeled by introducing its identity (or reference) into a specification in which this identity is not currently available (e.g. adding the identity to a specified set of object identities). Similarly, object destruction can be modeled by removing the identity of an object from a specification (e.g. removing the identity from a specified set) (Smith, 2005). Since object creation and destruction are very often used operations especially in dynamic structures, it is assumed that there is a unique global pool that stores the references to the created objects in the system:

$$\mid\ gObjectPool : \mathbb{P}\,\mathbb{O}$$

and every object in the system has a secondary variable ρ that refers to this pool. So from now on, the root class *Object* is defined as

Object

Δ
$\tau : \mathbb{T}$
$\rho : \mathbb{P}\,\mathbb{O}$

$\rho = gObjectPool$

and object creation can be modeled as an abbreviation:

$$\begin{aligned}\text{new}(c_1, c_2, ..., c_n, C) == (&\{c_1, c_2, ..., c_n\} \subset C \land \\ &\{c_1, c_2, ..., c_n\} \cap \rho = \varnothing \land \\ &\rho' = \rho \cup \{c_1, c_2, ..., c_n\})\end{aligned}$$

It declares n number of objects, $c_1, c_2, ..., c_n$, of the class C, which are not currently in *gObjectPool*. It also adds these objects to *gObjectPool* such that they will not be created again until they are removed from *gObjectPool*. The object destruction can also modeled as

$$\text{delete}(c_1, c_2, ..., c_n) == (\rho' = \rho \setminus \{c_1, c_2, ..., c_n\})$$

that removes these objects from the global pool.

The class *ActionServiceProxy* makes the operations Op_i ($i \in ActionID$) visible to its clients. These operations accepts the parameter lists as the corresponding operations in the class *ActionService* do, and accepts a time parameter as well for specifying the planned execution time. *ActionServiceProxy* has references to the related *ActionService*, and two active processes *ExecutionQueue* and *Scheduler* that work together to maintain and execute the actions at specified times:

ActionServiceProxy

$\upharpoonright((\text{INIT}, as, eq, s), Op_1, Op_2, \ldots, Op_n)$

$as : \downarrow ActionService$
$eq : ExecutionQueue_{©}$
$s : Scheduler$

INIT

$eq.\text{INIT}$

The attribute *eq* is a “contained” ($_{©}$) object of the type *ExecutionQueue*, which means that no two *ActionServiceProxy* can have the same *ExecutionQueue* object. This ensures the execution queue is only used by the containing object. The *ExecutionQueue eq* is initialized with an empty queue. However the *Scheduler s* is not required to be initialized because as said, *s* might be shared and could have been initialized by other objects.

Although the initial state predicate and the state variables as, eq, s are also visible to in the environment of an object of *ActionServiceProxy*, they should be "protected" from the objects outside the "package" of the pattern or the objects outside the inheritance tree. The clients of Timed Action pattern should only be able to access the operations $Op_1, Op_2, \ldots, Op_n$. Object-Z's visibility list defines the "public" members that are visible to any objects in the system and the ones not shown in the list will be "private" or hidden. However, it does not have facilities to specify such "protected" members in a "package" or in the inheritance tree as in UML and many object-oriented programming languages (for example, Java, C++, C#). To compensate, these "protected" members are parenthesized in the visibility list.

A local abbreviation $NewTimedAction(I)$ is introduced to simplify the specification. $NewTimedAction(I)$ is a Factory Method (Gamma et al., 1995; Metsker, 2002), which creates a new *TimedAction* object $ta!$ and a new *TentativeResult* object $tr!$:

$$NewTimedAction(I) ==$$

$$\begin{array}{|l}
\hline
\quad NewTimedAction_I \\
t? : \mathbb{T}_{\circledcirc} \\
p? : Dictionary \\
ta! : TimedAction \\
tr! : TentativeResult \\
\hline
\mathrm{new}(ta!, TimedAction) \land \mathrm{new}(tr!, TentativeResult) \\
\mathrm{new}(ta!.a, Action(I)) \land ta!.a.\textsc{Init} \\
ta!.a.as = as \land ta!.a.eq = eq \land ta!.a.p = p? \land ta!.a.tr = tr! \\
\textbf{if}\ t? = \mathsf{null}\ \textbf{then}\ ta!.t = \tau\ \textbf{else}\ ta!.t = t? \\
\hline
\end{array}$$

$NewTimedAction(I)$ accepts parameters $p? : Dictionary$ and a time value $t? : \mathbb{T}_{\circledcirc}$ as input. The input $p?$ and $t?$ as well as the reference to the *ActionService* Object and the *ExecutionQueue* object are stored in $ta!.a$. It associates an object of the type $Action(I)$ to the created *TimedAction* object ($ta!.a \in Action(I)$).

A new type is introduced that extends the type $\mathbb{T}$ with an undefined value null:

$$\mathbb{T}_{\circledcirc} ::= \mathbb{T} \mid \mathsf{null}$$

There is often such a situation that a null symbol is convenient to describe that a value is not defined and that this value is different from any value of a given type X. A generic free type is introduced to include such a null value

$$X_{\circledcirc} ::= X \mid \mathsf{null}$$

for a given type X.

If $t?$ is undefined, i.e., has a value of null, it is then converted to the current absolute time upon the creation of the new *TimedAction* object $ta!$. It allows the clients to request an action to be executed as soon as possible rather than at a defined absolute time:

$$\textbf{if}\ t? = \mathsf{null}\ \textbf{then}\ ta!.t = \tau\ \textbf{else}\ ta!.t = t?$$

Upon creation, the $Action(I)$ object $ta!.a$ also initiates an *TentativeResult* object $ta!.a.tr$ that is exposed to the environment as an output.

$$
\begin{aligned}
&Op_1 \mathrel{\widehat{=}} NewTimedAction(1)[o!/ta!] \mathbin{⨾} s.Subscribe\\
&Op_2 \mathrel{\widehat{=}} NewTimedAction(2)[o!/ta!] \mathbin{⨾} s.Subscribe\\
&\ldots\\
&Op_n \mathrel{\widehat{=}} NewTimedAction(n)[o!/ta!] \mathbin{⨾} s.Subscribe
\end{aligned}
$$

In each Op_i ($i \in ActionID$), I in $NewTimedAction(I)$ is replaced by i to form a concrete operation that takes input $p?$ and $t?$ from its client. It creates a $ta! : TimedAction$ where $ta!.a$ is of the type $Action(i)$. The created $ta! : TimedAction$ is then renamed to $o!$ as the input for the *Subscribe* operation of the object $s : Scheduler$. The *TentativeResult* output $tr!$ is passed through and output to the invoking process.

E.7 *TentativeResult*

The *TentativeResult* uses the Future pattern (Grand, 2002). At the architecture level together with the completion mechanism of asynchronous tasks in other components, it also falls into the Proactor pattern (Schmidt et al., 2000), in which the *TentativeResult* object acts as an event completion token for the clients. It allows a client to obtain the result of an action after the *Action* is executed at the planned time:

TentativeResult

Observable

$r : Dictionary$ [Results]
$tentative : \mathbb{B}$

$\textsc{Init} \mathrel{\widehat{=}} [\, tentative \,]$

$Complete \mathrel{\widehat{=}} r := r? \land tentative := \text{false} \mathbin{⨾} Notify$

When a client requests an action (Op_i, $i \in ActionID$) through an *ActionServiceProxy*, a *TentativeResult* is produced immediately for the client, in which $r : Dictionary$ is reserved in $Action(i)$ to store its results. In this specification, TentativeResult is defined as an *Observable* (see *Observable* on page 62), so that the clients may register themselves to the *TentativeResult* as *Observer*s in order to get notified when the *TimedAction* completes the action execution and makes the result ready. Or, instead of using the Observer pattern, a client may also "rendezvous" (Grand, 2002, p. 541) with the *TentatvieResult* to get the result by either blocking or polling until it is no longer "tentative".

BACKGROUND MATERIAL F

Specifications of the Synchronizable Object Pattern

This background material presents the formal specification of the components of the Synchronizable Object pattern that is introduced in section 6.2 of chapter 6.

F.1 Synchronization states

Let's first prepare some concepts used by *Synchronizable* objects. A synchronizable object is a simple finite state machine with four states stopped, ready, started, paused as shown in figure F.1 on the next page. The initial state is stopped. A synchronizable object must get ready before it can be started. A ready state is necessary to model many multimedia objects that requires computational resources to be allocated and certain amount of data to be prefetched before it can be started immediately. Once it has been started, it can be paused and be restarted again.

In some multimedia systems, for example in PREMO (Duke et al., 1999; Herman et al., 1998), similar objects are used for synchronization, but these objects also have a waiting state. The difference between waiting state and the paused state is that waiting can only be caused by an internal operation, otherwise they are the same state. The design here prefers a simplified state machine without a waiting state.

A free type *SyncState* is introduced to identify these states:

$$\mathit{SyncState} ::= \mathsf{stopped} \mid \mathsf{ready} \mid \mathsf{started} \mid \mathsf{paused}$$

F.2 Coordinates

A synchronizable object controls the position and progress along its own coordinates. Different synchronizable objects may use different coordinates. For example, a video stream might use frame numbers as coordinates, and a clock might simply take $\mathbb{T}$ as

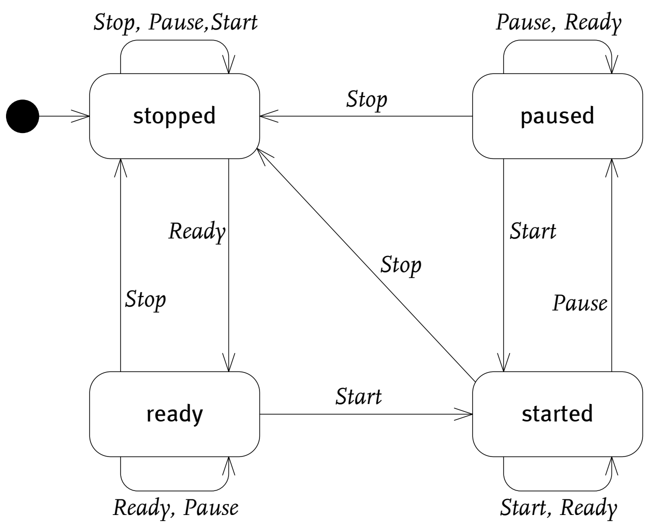


Figure F.1: State transitions of a synchronizable object

its coordinates. A synchronizable object may have a function to map the coordinates to time and time constraints can be added to the operations – however the coordinates are not necessarily always time related, for example, a TTS engine may take white spaces or punctuation locations between words or sentences as coordinates. The concept of coordinates is more general than the time. To model the progression steps, the coordinates $\mathbb{C}$ is introduced

$$[\mathbb{C}]$$

and defined as a discrete set of *points* that can be ordered with a relation $<$ (also see figure F.2 on the facing page):

$$\_ < \_ : \mathbb{C} \leftrightarrow \mathbb{C}$$

$$\begin{array}{ll}
\forall c : \mathbb{C} \bullet \neg (c < c) & [\,\textsf{Irreflective}] \\
\forall c_1, c_2, c_3 : \mathbb{C} \bullet c_1 < c_2 \land c_2 < c_3 \Rightarrow c_1 < c_3 & [\,\textsf{Transitive}] \\
\forall c_1, c_2 : \mathbb{C} \bullet c_1 < c_2 \Rightarrow \neg (c_2 < c_1) & [\,\textsf{Antisymmetric}] \\
\forall c_1, c_2 : \mathbb{C} \bullet c_1 < c_2 \lor c_1 = c_2 \lor c_2 < c_1 & [\,\textsf{total}]
\end{array}$$

The coordinates $\mathbb{C}$ is discrete, which means every coordinate has a “closest” coordinate on each side, unless there is no coordinate on that side:

$$\forall c_1, c_2 : \mathbb{C} \bullet c_1 < c_2 \Rightarrow \exists c_3 : \mathbb{C} \bullet c_1 < c_3 \land \nexists c_4 : \mathbb{C} \bullet c_1 < c_4 < c_3$$
$$\forall c_1, c_2 : \mathbb{C} \bullet c_1 < c_2 \Rightarrow \exists c_3 : \mathbb{C} \bullet c_3 < c_2 \land \nexists c_4 : \mathbb{C} \bullet c_3 < c_4 < c_2$$

F.3 Repeating positions

A client of a synchronizable object might want to traverse the coordinates more than once, or possibly traverse in an infinite loop, for example, repeating a multimedia presentation several times or repeating it forever. This means a given point within the coordinates may be visited multiple times. A new type is defined to combine a point in the coordinates with a visit number,

$$Position == \mathbb{C} \times \mathbb{N}$$

and a total order is defined over *Position*:

$$\begin{array}{|l} \_\, prec \,\_ : Position \leftrightarrow Position \\ \hline \forall c_1, c_2;\ n_1, n_2 \bullet \\ \qquad (c_1, n_1)\ prec\ (c_2, n_2) \Leftrightarrow n_1 < n_2 \lor (n_1 = n_2 \land c_1 < c_2) \end{array}$$

F.4 Progression directions

A synchronizable object may advance along its coordinates, or backward in an inverted direction:

$$Direction ::= \mathsf{forward} \mid \mathsf{backward}$$

Figure F.2 shows a subset of *Position*, which has a finite set of coordinates $\{c_1, c_2, \ldots, c_n\}$ to be visited 3 times in the direction of forward.

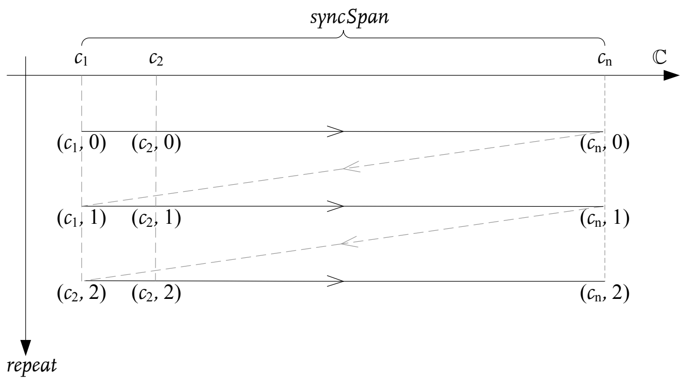


Figure F.2: A subset of *Position*

F.5 Synchronization Events

Some of the coordinates can be attached with synchronization events. *Event* is defined as the super class for all the events that the objects need to notify others:

$$\begin{array}{|l} \textit{Event} \\ \hline \boxed{source : \mathbb{O}_{\circledcirc}} \end{array}$$

$$SetSource \mathrel{\widehat{=}} [\,\Delta(source)\ s? : \mathbb{O}_{\circledcirc} \mid source' = s?\,]$$

which has an attribute *source* storing the reference to the originating object if desired. Other details are left to subclasses. To distinguish synchronization events from other events (for example, user input events), *SyncEvent* is defined as a subclass of *Event*:

$$\begin{array}{|l}
\textit{SyncEvent} \\
\hline
\textit{Event} \\
\hline
\end{array}$$

Subtyping can be used to identify different types of *SyncEvent* objects, and the subclasses may also add other state variables to carry extra information. Instead of giving all the possible different types in advance, which is often not feasible, subtyping is a more flexible and extensible solution for identifying objects (Simons, 2002). For example, if the transition between different *SyncState*s is interesting for other objects, a new class *StateSyncEvent* can be defined as:

$$\begin{array}{|l}
\textit{StateSyncEvent} \\
\hline
\textit{SyncEvent} \\
\\
\begin{array}{|l}
\hline
oldState, newState : SyncState \\
\hline
oldState \neq newState \\
\hline
\end{array} \\
\hline
\end{array}$$

and an object $se : \downarrow SyncEvent$ can be identified as a *StateSyncEvent* object if $se \in StateSyncEvent$ is true.

F.6 *Synchronizable*

Let's first give a visibility list that declares the visible state variables and operations:

$$\begin{array}{|l}
\textit{Synchronizable} \\
\hline
\upharpoonright(\ eventDispatcher, repeatNumber, repeatCount, syncState, begin, end, \\
\quad current, syncElements, direction, \qquad\qquad [\text{Visible state variables}] \\
\quad \text{I}{\small NIT}, Start, TryStart, Stop, Pause, Visit, Step, \\
\quad setRepeatNumber, setDirection, setBegin, setEnd, setBeginEnd, \\
\quad attachSyncEvent, detachSyncEvent, addSyncElements, \\
\quad removeSyncElements \qquad\qquad [\text{Visible operations}] \\
\ )
\end{array}$$

State variables The state schema is then defined as follows:

Readable state variables:

$eventDispatcher : \downarrow EventDispatcher$

$repeatNumber : \mathbb{N}$ [How many times to repeat]

$repeatCount : \mathbb{N}$ [How many times repeating completed]

$syncState : SyncState$ [Current synchronization state]

$syncElements : \mathbb{C} \leftrightarrow \downarrow SyncEvent$ [Synchronization elements]

$begin : \mathbb{C}_{\circledcirc}$ [Where to begin]

$end : \mathbb{C}_{\circledcirc}$ [Where to end]

$current : \mathbb{C}_{\circledcirc}$ [Current location in the coordinates]

$syncSpan : \mathbb{P}\,\mathbb{C}$ [Synchronization span between *begin* and *end*]

$direction : Direction$ [traversal direction]

Local state variables that are invisible to others:

$positions : \mathbb{P}\, Position$ [Positions to be traversed]

$curPosition : Position_{\circledcirc}$ [Current position]

$lastPosition : Position_{\circledcirc}$ [The position visited last time]

$\_ \prec \_ : Position \leftrightarrow Position$ [Order of traversal]

$syncSpan = (\mathbf{if}\, begin = \mathsf{null}\, \mathbf{then}\, \mathbb{C}\, \mathbf{else} \{c : \mathbb{C} \mid begin \leqslant c\}) \cap$
$\qquad (\mathbf{if}\, end = \mathsf{null}\, \mathbf{then}\, \mathbb{C}\, \mathbf{else} \{c : \mathbb{C} \mid c \leqslant end\})$

[*begin* and *end* define the synchronization span]

$positions = syncSpan \times (\mathbf{if}\, repeatNumber = 0\, \mathbf{then}\, \mathbb{N}$
$\qquad \mathbf{else} \{n : \mathbb{N} \mid n < repeatNumber\})$

[Positions are always defined by *syncSpan* and *repeatNumber*]

$\mathbf{if}\, curPosition = \mathsf{null}\, \mathbf{then}\, current = \mathsf{null} \wedge repeatCount = 0$
$\quad \mathbf{else}\, curPosition = current \mapsto repeatCount$

[Decompose current position to its coordinate and repeat count]

$direction = \mathsf{forward} \Rightarrow (\_ \prec \_) = (\_\, prec\, \_)$

$direction = \mathsf{backward} \Rightarrow (\_ \prec \_) = (\_\, prec^{\sim}\, \_)$

[Traversal direction defines the order of the positions]

The class aggregates an object of $\downarrow$*EventDispatcher*. An alternative could be to inherit the event handling mechanisms from the type $\downarrow$*EventDispatcher*. However doing so would interweave the synchronization process with the event handling process. Aggregation allows to separate them in independent processes and to dispatch the events asynchronously.

The attributes *repeatNumber* keeps the total number of the traversal loops. If *repeatNumber* is 0, the range of *postions* to be traversed is set to cover all integer numbers, which means the *Synchronizable* object will repeat forever if it is not interrupted.

The set *syncSpan* defines the span of the synchronization coordinates – The client of a *Synchronizable* object may define the boundaries to limit the synchronization to a subset of coordinates. This is an often required behavior for example when synchronizing a multimedia object, where the client might specify the *begin* and the *end* time within the actual duration of the object.

The relation *syncElements* attaches *SyncEvent* objects to certain coordinates. Note that one coordinate may have multiple *SyncEvent* objects attached.

The attribute *positions* defines the positions (pairs of coordinates in *syncSpan* and the repeat counter) that will be passed during traversal. The attributes *curPosition* and *lastPosition* keeps track of the current traversal position, and the position that has just visited last time. The relation $\prec$ defines the order of the traversal according to the specified *direction*. The attributes *positions*, *curPosition* and *lastPosition* are for internal use and hence they are not visible to other objects.

Initial state Upon initialization, The repeat number by default is 1 and the repeat count starts from 0. The synchronization state is set to be stopped and there are no *syncEvent* attached to any coordinates. The *syncSpan* covers all the possible coordinates and the traverse direction is forward. The attribute *curPosition* and *lastPosition* are set to null, which means there is no last visited position, and the current position is not given yet.

$$
\begin{array}{|l}
\textit{INIT} \\
\hline
\mathit{repeatNumber} = 1 \land \mathit{repeatCount} = 0 \\
\mathit{direction} = \mathsf{forward} \land \mathit{syncSpan} = \mathbb{C} \\
\mathit{syncState} = \mathsf{stopped} \land \mathit{syncElements} = \varnothing \\
\mathit{curPosition} = \mathsf{null} \land \mathit{lastPosition} = \mathsf{null} \\
\hline
\end{array}
$$

Next the operations are going to be defined.

Synchronization operations Many operations may cause the synchronization state change and trigger corresponding *StateSyncEvent*. A schema *Transit* is defined to catch this common behavior:

$$
\begin{array}{|l}
\mathit{Transit}_0 \\
\hline
\Delta(\mathit{syncState}, \mathit{syncState}') \\
e! : \mathit{StateSyncEvent}_{\circledcirc} \\
\hline
\mathbf{if}\ \mathit{syncState} \neq \mathit{syncState}' \\
\quad \mathbf{then}\ e! \neq \mathsf{null} \land e!.\mathit{source} = \mathit{self} \land \\
\quad\quad e!.\mathit{oldState} = \mathit{syncState} \land e!.\mathit{newState} = \mathit{syncState}' \\
\quad \mathbf{else}\ e! = \mathsf{null} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{Transit} \mathrel{\widehat=} \mathit{Transit}_0 \mathbin{⨾} ([\, e? : \mathit{StateSyncEvent}_{\circledcirc} \,] \bullet \\
\qquad (\mathbf{if}\ e? \neq \mathsf{null}\ \mathbf{then}\ \mathit{eventDispatcher}.\mathit{Dispatch}))
\end{array}
$$

The operation $\mathit{Transit}_0$ outputs a well defined *StateSyncEvent* with its *source* pointing to the producing object if the state transition happens, otherwise it outputs a null value. The operation *Transit* picks up the output from $\mathit{Transit}_0$, passes it to the operation *Dispatch* of *eventDispatcher*, or simply consumes the output if $e?$ is null (the anonymous empty operation [ ] does nothing).

For easy reading, the **if**... **then**... **else**... operator from standard Z is extended here for Object-Z operation composition, which is equivalent to an abbreviation

$$(\textbf{if}\ p\ \textbf{then}\ Op_1\ \textbf{else}\ Op_2) == ([\,p\,] \land Op_1 \mathbin{[\!]} [\,\neg\, p\,] \land Op_2)$$

when it is used for operation compositions. Further, if Op_2 is an empty operation, let's write

$$\textbf{if}\ p\ \textbf{then}\ Op_1$$

instead of

$$\textbf{if}\ p\ \textbf{then}\ Op_1\ \textbf{else}\ [\ ]$$

The operations that manipulate the synchronization state are defined next. The operation *Stop* puts the synchronizable object into the stopped state from any other state. Stopping an object also causes its repeat counter, the current position and the last visited position to be reset to their initial state.

$$\begin{array}{|l}
\quad Stop_0 \\
\hline
\Delta(syncState, repeatCount, curPosition, lastPosition) \\
\hline
syncState' = \mathsf{stopped} \land repeatCount' = 0 \\
curPosition' = \mathsf{null} \land lastPosition' = \mathsf{null} \\
\hline
\end{array}$$

$$Stop \mathrel{\widehat=} Stop_0 \land Transit$$

The operation *Ready* gets the stopped synchronizable object ready for starting. This operation has no effects if the object is in any state other than stopped:

$$\begin{array}{|l}
\quad Ready_0 \\
\hline
\Delta(syncState) \\
\hline
syncState = \mathsf{stopped} \Rightarrow syncState' = \mathsf{ready} \\
syncState \in \{\mathsf{started}, \mathsf{paused}, \mathsf{ready}\} \Rightarrow syncState' = syncState \\
\hline
\end{array}$$

$$Ready \mathrel{\widehat=} Ready_0 \land Transit$$

The operation *Ready* tends to be extended when defining a multimedia object , which will possibly engage more complicated prefetching behavior than just changing the state to ready as it is shown here.

A synchronizable object can be started when it is ready or paused. However, to start from ready, the object must have a position to start from, which requires that either the *begin* location is given, or there exists the *minimum*[1]in the coordinate system. The operation has no effects if the object has already been started, or if it is stopped:

[1] Let's define *minimum* as a total function that finds the minimum element of a set X with regard to a relation r as follows:

$$minimum[X] == \lambda\, r : X \leftrightarrow X;\ a : \mathbb{P}\, X \bullet (x : X_{\circledcirc} \mid \\ \textbf{if}\ lowerBound(r, a) = \varnothing\ \textbf{then}\ x = \mathsf{null}\ \textbf{else}\ \{x\} = a \cap lowerBound(r, a))$$

$$\begin{array}{|l}
\textit{Start}_0 \\
\hline
\Delta(\textit{syncState}, \textit{curPosition}) \\
\hline
\textit{syncState} = \mathsf{ready} \Rightarrow \\
\quad \textit{curPosition}' = \textit{minimum}(\_ \prec \_, \textit{positions}) \\
\quad \textbf{if}\ \textit{curPosition}' = \mathsf{null}\ \textbf{then}\ \textit{syncState}' = \mathsf{stopped} \\
\quad\quad \textbf{else}\ \textit{syncState}' = \mathsf{started} \\
\textit{syncState} = \mathsf{paused} \Rightarrow \textit{syncState}' = \mathsf{started} \\
\textit{syncState} = \mathsf{started} \Rightarrow \textit{syncState}' = \textit{syncState} \\
\hline
\end{array}$$

$$\textit{Start} \mathrel{\widehat=} \textit{Start}_0 \wedge \textit{Transit}$$

For a stopped object to start, it must get *ready* first. The operation *TryStart* tries to start the synchronizable object from any states:

$$\textit{TryStart} \mathrel{\widehat=} \textbf{if}\ \textit{syncState} = \mathsf{stopped}\ \textbf{then}\ \textit{Ready} \mathbin{⨟} \textit{Start}\ \textbf{else}\ \textit{Start}$$

Once the synchronizable object has been started, it can be put in to the paused by the operation *Pause* defined below. This operation does not have any effect if the object is in any synchronization states other than started:

$$\begin{array}{|l}
\textit{Pause}_0 \\
\hline
\Delta(\textit{syncState}) \\
\hline
\textit{syncState} = \mathsf{started} \Rightarrow \textit{syncState}' = \mathsf{paused} \\
\textit{syncState} \in \{\mathsf{stopped}, \mathsf{ready}, \mathsf{paused}\} \Rightarrow \textit{syncState}' = \textit{syncState} \\
\hline
\end{array}$$

$$\textit{Pause} \mathrel{\widehat=} \textit{Pause}_0 \wedge \textit{Transit}$$

Position operations The following operations are so called "setter" operations that change the value of the state variables explicitly. These operations are straightforward, nonetheless it should be noticed that because the postcondition of every operation must yield to the predicates in the state schema, changing values of the state variables may implicitly change the value of the others. For example a new *begin* value will result in a new *syncSpan* and in turn a new set of *positions*, and a new *direction* will also result in new meaning of $\prec$.

$$\begin{aligned}
\textit{SetRepeatNumber} \mathrel{\widehat=}\ & [\textit{syncState} = \mathsf{stopped}] \wedge \\
& [n? : \mathbb{N}] \bullet \textit{repeatNumber} := n? \\
\textit{SetDirection} \mathrel{\widehat=}\ & [\textit{syncState} = \mathsf{stopped}] \wedge \\
& [d? : \textit{Direction}] \bullet \textit{direction} := d?
\end{aligned}$$

where the total function

$$\textit{lowerBound}[X, Y] == \lambda\, r : X \leftrightarrow Y;\ b : \mathbb{P}\, Y \bullet \{x : X \mid \forall\, y : b \bullet x \mapsto y \in r\}$$

defines the lower bound of a set *b* through a relation *r*, as the set of those elements of the source type of *r* that are related to all elements in *b* by *r*.

It is easy to prove that the minimum, if any, is unique if the relation is antisymmetric.

The *syncSpan* and the *positions* can be changed by setting the minimum element of the synchronization coordinates, or the maximum element, or both at the same time:

$$\begin{array}{l}
SetBegin \mathrel{\widehat{=}} [\, syncState = \mathsf{stopped} \,] \land [\, begin? : \mathbb{C}_{\circledcirc} \,] \bullet begin := begin? \\
SetEnd \mathrel{\widehat{=}} [\, syncState = \mathsf{stopped} \,] \land [\, end? : \mathbb{C}_{\circledcirc} \,] \bullet end := end? \\
SetBeginEnd \mathrel{\widehat{=}} SetBegin \land SetEnd
\end{array}$$

All these "setter" operations are not allowed to take place when the object is in any other states except the stopped state, to avoid semantic confusion. It is however technically possible to enable these operations in all other states, but then the developer has to take care of the state variables *positions*, *curPosition* and *lastPosition* so that they are not conflicting with each other, and so that the resulting state should be easily understandable.

Event operations The operation *SetEventDispatcher* associates an event dispatcher to the synchronizable object:

$$SetEventDispatcher \mathrel{\widehat{=}} [\, ed? : EventDispatcher \,] \bullet eventDispatcher := ed?$$

The next two operations manipulate the synchronization events by attaching a *SyncEvent* object to a coordinate or detaching it:

$$\begin{array}{|l}
\hline
AttachSyncEvent \\
\Delta(syncElements) \\
c? : \mathbb{C} \\
e? : \downarrow SyncEvent \\
\hline
syncElements' = \\
\quad syncElements \cup \{c? \mapsto e?\} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
DetachSyncEvent \\
\Delta(syncElements) \\
c? : \mathbb{C} \\
e? : \downarrow SyncEvent \\
\hline
syncElements' = \\
\quad syncElements \setminus \{c? \mapsto e?\} \\
\hline
\end{array}$$

It can also be desirable to have the operations that add or remove a set of synchronization elements all together, for example in the case of setting up a "periodic" synchronization behavior:

$$\begin{array}{|l}
\hline
AddSyncElements \\
\Delta(syncElements) \\
ses? : \mathbb{C} \leftrightarrow \downarrow SyncEvent \\
\hline
syncElements' = syncElements \cup ses? \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
RemoveSyncElements \\
\Delta(syncElements) \\
ses? : \mathbb{C} \leftrightarrow \downarrow SyncEvent \\
\hline
syncElements' = syncElements \setminus ses? \\
\hline
\end{array}$$

Once a position is being visited, all the events attached to the coordinates from the last visited position (excluded) to the current position (included) should be triggered:

$$
\begin{array}{l}
\textit{GetEventsQueue} \\
\hline
eq! : \operatorname{seq} \mathbb{P}\, SyncEvent \\
pq : \operatorname{seq} Position \\
\hline
pq = sort(\_ \prec \_, \\
\quad \operatorname{dom}(syncElements) \lhd \\
\qquad (\{curPosition\} \cup \\
\qquad\ \mathbf{if}\ lastPosition = \mathsf{null}\ \mathbf{then}\ \varnothing \\
\qquad\qquad \mathbf{else} \{p : Position \mid lastPosition \prec p \prec curPosition\} \\
\qquad) \\
\quad) \\
\# eq! = \# pq \\
\forall\, i : 1 \mathinner{..} \# eq! \bullet eq!(i) = \\
\quad \{se : syncElements \mid \operatorname{first}(se) = \operatorname{first}(pq(i)) \bullet \operatorname{second}(se)\}
\end{array}
$$

Although the positions after the last visited position and before the current position might not have been visited, the events attached to these positions should not be ignored. The operation *GetEventsQueue* first finds out all the related positions which have coordinates attached with synchronization events, sort[2] them into a sequence, then output another sequence of *SyncEvent* sets in the order of the position sequence. The output is serialized into a sequence instead of a set, because the order information is important for the following operation *DispatchEventsQueue*, so that the synchronization events can be triggered sequentially along the positions. The events attached to a position before other positions may have their semantic effect to happen first. Also notice that the events attached to a coordinate can be included more than once at different positions because of repeating.

The operation *DispatchEventsQueue* takes a sequence of *SyncEvent* sets as input, sends the head of the sequence to the operation *DispatchSyncEvents*, then directs the tail of the sequence back to the operation *DispatchEventsQueue* until the input sequence is empty, using a recursive definition:

$$
\begin{array}{l}
DispatchEventsQueue \mathrel{\widehat{=}} \big[\, eq? : \operatorname{seq} \mathbb{P}\, SyncEvent \,\big] \bullet \\
\quad \mathbf{if}\ eq? \neq \langle\,\rangle\ \mathbf{then}(DispatchSyncEvents(\operatorname{head}(eq?)) \mathbin{;} \\
\qquad\qquad\qquad\qquad DispatchEventsQueue(\operatorname{tail}(eq?)))
\end{array}
$$

where the operation *DispatchSyncEvents* simultaneously sends the events to the event handler for dispatching:

$$
\begin{array}{l}
DispatchSyncEvents \mathrel{\widehat{=}} \big[\, se? : \mathbb{P}\, SyncEvent \,\big] \bullet \\
\qquad\qquad\qquad \bigwedge e : se? \bullet eventDispatcher.Dispatch(e)
\end{array}
$$

[2] The total function *sort* maps a finite subset of a given type X to a sequence according to a given relation r, by *squash*ing (Spivey, 1992, p.121) any bijective function that maps a set of nature numbers to the given finite subset:

$$
\begin{array}{l}
sort[X] == \lambda\, r : X \leftrightarrow X;\ a : \mathbb{F}\, X \bullet \\
\qquad \forall f : \mathbb{N}_1 \rightarrowtail\!\!\!\!\twoheadrightarrow pos \mid \forall\, n_1, n_2 : \operatorname{dom}(f) \bullet n_1 < n_2 \Leftrightarrow f(n_1) \mapsto f(n_2) \in r \bullet squash(f)
\end{array}
$$

Progression operations Having defined the event operations, it is ready to give "visiting a position" a preliminary meaning:

$$
\begin{array}{|l}
Visit \mathrel{\widehat=} [\, syncState = \mathsf{started} \land curPosition \neq \mathsf{null} \land \\
\qquad curPosition \neq lastPosition \,] \land \\
\qquad (GetEventsQueue \mathbin{⨾} DispatchEventsQueue)
\end{array}
$$

which does noting but dispatching the events. This operation tends to be extended or overwritten to give the actual meaning in a concrete synchronizable object. Depending on the type of the synchronizable objects, while visiting the object it may for example perform data presentation for any data identified by the current coordinate, or invoke extra operations that take attached events as input before dispatching them.

Once the current position has been visited, the following *CurToLast* operation should be invoked to record the current position before visiting the next position:

$$
\begin{array}{|l}
CurToLast \mathrel{\widehat=} [\, \Delta(lastPosition) \mid lastPosition' = curPosition \,]
\end{array}
$$

Besides event handling, so far the progression of a synchronizable object is modeled by specifying the coordinates as milestones, unfolding repetition into positions, and defining the visiting order as per direction. The object has now the static basis for dynamic progression. However, the dynamic behavior, that is, how the synchronized object moves from one position to another, is not yet clear.

A default "stepping" behavior is described as follows, assuming the next position to visit is the closest position after the current one, without knowing whether it is driven by its own process or by anything else:

$$
\begin{array}{|l}
\underline{\quad NextVisit \quad} \\
\Delta(curPosition) \\
\hline
curPosition \neq \mathsf{null} \\
curPosition' = minimum(\_ \prec \_, \{p : positions \mid curPosition \prec p\}) \\
\hline
Step \mathrel{\widehat=} \mathbf{if}\ syncState = \mathsf{started} \land curPosition = \mathsf{null}\ \mathbf{then}\ Stop \\
\qquad \mathbf{else}(Visit \parallel CurToLast \parallel NextVisit)
\end{array}
$$

For the operation *Step* to succeed, the current position must be visitable (see the conjoined conditions on the operation *Visit* on this page). If the current position is set to null, there is no way to find the immediate next and it will *Stop* by itself. This may happen when there is no next position to visit. The parallel composition operator $\parallel$ allows concurrent component operations, but also enables inter-object communication between the components by equating any input to one of the components of $\parallel$ with any output from the other component that has the same base name. When there is no inter-object communication, it behaves like the conjunction operator. However Conjoining an inherited operation in the inheriting class does not affect the meaning of other inherited operations which are defined in terms of this operation (Smith, 2000, p.47), whereas the parallel operator does not have this constraint.

Visiting a position, storing the current position to *lastPosition* and finding out the position to visit next should be done simultaneously. It can also be done sequentially to fulfill almost the same functionality, but the parallel composition means that the next position must be found while visiting the current position, so that there are no extra activities between two visits in sequential steps ($Step \mathbin{⨾} Step$), to achieve seamless and continuous visiting.

This finishes the definition of the abstract specification of a synchronizable object. It is defined as such so that no media specific semantics is directly attached to it. For example, there is no notion of time although it is crucial for time-based media objects. Subclasses, realizing specific media control should, through specification, attach concrete semantics to the object through their choice of the type of the internal coordinate system, through a proper specification of what "visiting a position" means, and through a proper specification of how the object should move from the current position to the next.

F.7 *ActiveSynchronizable*

Some synchronizable objects can be self-driven – it has its own process to step forward:

ActiveSynchronizable

$\upharpoonright(\cdots)$ [Visibility list same as in *Syncronizable*]

Synchronizable

IdleTick

$\neg\ \mathrm{pre}(Step)$

$\tau' = \tau + 1$

$\textsc{Process} \mathrel{\widehat=} \mu\, AS \bullet (Step \mathbin{[]} IdleTick) \mathbin{⨾} AS$

The visibility list in Object-Z is not inherited by the subtypes (Smith, 2000). It is however trivial to include the same list from a super class. Let's write $\upharpoonright(\cdots)$ to include all the visible state variables and operations from the super classes to the visibility list.

A good example of *ActiveSynchronizable* is a piece of text being rendered by a TTS engine, where white spaces and punctuation locations can be used as coordinates that separate the text to short sentences. The engine then renders the text sentence by sentence. There are no time constrains – how long it will take to render the text depends on the rendering configurations. But once started, the engine should render the sentences one after another automatically.

F.8 *SyncMediaObject*

The example of text being rendered by a TTS engine is not only *Synchronizable*, but also a *MediaObject*. A media object is to be consumed (for example, read, heard, viewed), hence it has to "present" something from somewhere in physical means (for example lights, sounds, images and moving pictures), so that the users can in some way receive it into their sensory systems. First three special types are introduced:

$$[\mathbb{D}, \mathbb{R}]$$

and

$$\begin{array}{|l}
\hline \mathbb{S} \\
\textit{String} \\
\hline
\end{array}$$

where $\mathbb{D}$ represents "something" – the data in the digital world, and $\mathbb{S}$ is a special type of string that represents "somewhere" – the source of the data. "In physical means" is model as presenting data to the "real physical world" $\mathbb{R}$.

To separate the concern, these additional attributes and behaviors are modeled as of type *MediaObject*:

$$\begin{array}{|l}
\hline \textit{MediaObject} \\
\quad \boxed{\textit{src} : \mathbb{S}} \\
\textit{Retrieve} \mathrel{\widehat{=}} \left[c? : \mathbb{C};\ d! : \mathbb{D}_{\circledcirc} \right] \\
\textit{Present} \mathrel{\widehat{=}} \left[d? : \mathbb{D}_{\circledcirc};\ r : \mathbb{R};\ pr : \mathbb{D} \rightarrow \mathbb{R} \mid d? \neq \mathsf{null} \Rightarrow r = pr(d?) \right] \\
\hline
\end{array}$$

The operation *Retrieve*, which takes a coordinate as the parameter, retrieves the corresponding data from the source as output. A *MediaObject* is able to *Retrieve* needed data, and has an operation to *Present* the data into something physical. Note that the operation *Retrieve* may fail to get the required data, either because there is no data associated with the coordinate $c?$, or because the data can not be available in time during retrieving. In either case, *Retrieve* outputs a null value and *Present* does nothing.

A *SyncMediaObject* is then a *Synchronizable MediaObject* which retrieves and presents the data while visiting the *current* position:

$$\begin{array}{|l}
\hline \textit{SyncMediaObject} \\
\upharpoonright(\cdots) \\
\textit{Synchronizable}[\textit{Visit}_s / \textit{Visit}] \\
\textit{MediaObject} \\
\textit{Visit} = \textit{Visit}_s \parallel \\
\qquad (\textit{Retrieve}(\textit{current}) \parallel \\
\qquad\ \left[d? : \mathbb{D}_{\circledcirc} \right] \bullet \mathbf{if}\, d? = \mathsf{eof}\, \mathbf{then}\, \textit{Stop}\, \mathbf{else}\, \textit{Present}) \\
\hline
\end{array}$$

The source of the media object may explicitly send a signal of the "end of file" $(\langle\!\langle eof \rangle\!\rangle : \mathbb{D})$ to indicate that there is no further data available for presentation. In this case the operation *Stop* is invoked.

F.9 *ActiveSyncMediaObject*

Therefore, a TTS media object should inherit both the active synchronization behaviors from the type *ActiveSynchronizable* and the data operations from the type *MediaObject*. It can be defined as of the type *ActiveSyncMediaObject*:

$$
\begin{array}{|l}
\textit{ActiveSyncMediaObject} \\
\hline
\upharpoonright(\cdots) \\
\textit{ActiveSynchronizable} \\
\textit{MediaObject} \\
\textit{Visit} = \textit{Visit}_s \;\; \| \\
\qquad\quad (\textit{Retrieve}(\textit{current}) \;\; \| \\
\qquad\quad\; [\, d? : \mathbb{D}_{\circledcirc} \,] \bullet \textbf{if}\; d? = \mathsf{eof}\; \textbf{then}\; \textit{Stop}\; \textbf{else}\; \textit{Present}) \\
\textsc{Process} \mathrel{\widehat=} \mu\, \textit{AMO} \bullet (\textit{Step} \mathrel{[]} \textit{IdleTick}) \mathbin{⨾} \textit{AMO} \\
\hline
\end{array}
$$

Since only inheriting the active type *ActiveSynchronizable* does not make *ActiveSyncMediaObject* automatically active, the operation $\textsc{Process}$ is explicitly defined.

F.10 *Timer*

It is now the time for a *Timer*. For time-based synchronization, such a *Timer* is often needed in order to add time constraints to operations. The type $\mathbb{T}$ is discrete and has the order relation $<$ defined (see required properties on page 278), therefore it can be naturally used as a coordinate system. A *Timer* is defined as a *Synchronizable* object with its coordinate system replaced by $\mathbb{T}$:

$$
\begin{array}{|l}
\textit{Timer} \\
\hline
\upharpoonright(\cdots) \\
\textit{ActiveSynchronizable}[\mathbb{T}/\mathbb{C}]
\end{array}
$$

A *Timer* object may advance faster or slower according to the request. It has a state variable *speed* that takes a non-zero real number as the speeding factor, and a function *timing* maps each position to the total time that should be used to advance from the first position to a given position at the given *speed*.

$$
\begin{array}{|l}
\textit{speed} : \mathbb{R} \\
\textit{timing} : \textit{Position} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{T} \\
\hline
\textit{speed} \neq 0 \\
\forall\, p : \textit{positions} \bullet \textit{timing}(p) = \\
\qquad\qquad \#\{p_1 : \textit{positions} \mid p \prec \textit{curPosition}\} \div \textit{speed} \\
\hline
\end{array}
$$

The function *timing* will be updated whenever a new set of *positions* is defined or a new *speed* is given. An extra operation *SetSpeed* is needed for the other objects to change the *speed*:

$SetSpeed \mathrel{\widehat=} [syncState = \mathsf{stopped}] \wedge [speed? : \mathbb{R}] \bullet speed := speed?$

Note that in the coordinate system of a *Timer* object, the distance between two successive time coordinates is 1 time unit, hence the total time that should be used to move from one position to another can be calculated by simply counting the number of positions in between. Speeding requires this total time to be divided by the speed factor – faster means less time should be used.

The crucial part of a *Timer* object is its *NextVisit* operation:

```
NextVisit
  $\Delta(curPosition)$
  $p_{min}, p_{max} : Position_{\circledcirc}$
  ----------
  $p_{min} = minimum(\_ \prec \_,$
      $\{p : positions \mid \tau' - \tau = timing(p) - timing(curPosition)\})$
  $p_{max} = maximum(\_ \prec \_, positions)$
  **if** $p_{min} \neq \mathsf{null}$ **then** $curPosition' = p_{min}$
    **else if** $p_{max} \neq \mathsf{null} \wedge p_{max} \neq curPosition$
      **then** $curPosition' = p_{max}$ **else** $curPosition' = \mathsf{null}$
```

The operation *NextVisit* first tries to find such a position that

- the total time spent from *curPosition* to this position

  $$timing(p) - timing(curPosition)$$

  is enough to finish the current visit,

- the actual time used ($\tau' - \tau$) for visiting current position can be more than the necessary minimum, but must be equal to this total time.

- it should be as close as possible to *curPosition*.

If such a position does not exist and the *maximum*[3] position, if exists, has not been visited, it means that the time needed to finish the current visit is beyond the time distance between *curPosition* and the last position. The last position should however not be ignored, otherwise the events attached to the positions after *curPosition* would never be dispatched. In this case, the operation *NextVist* forces the last position to be visit next. If all of the above fails, The operation sets the next position to null.

A *Timer* object has a process of its own to move forward:

$\textsc{Process} \mathrel{\widehat=} \mu\, T \bullet (Step \mathbin{[]} IdleTick) \mathbin{\fatsemi} T$

[3] The total function *maximum* on a set X with regard to a relation r is a "dual" function of *minimum* (see footnote on page 283):

$$maximum(r, a) == \lambda\, r : X \leftrightarrow X;\ a : \mathbb{P}\,X \bullet minimum(r^{\sim}, a)$$

F.11 *TimedSynchronizable*

Many media objects, for example MP3 audio and MPEG-4 video, are time based. These objects have their own coordinate systems, for example, frame numbers in a video stream. However when playing back, every frame has to be presented in a fixed or variable rate, which often is referred as the framerate (for video streams) or the bitrate (for audio streams). There is a mapping between their own coordinate systems and the require presentation time.

There are two ways to establish such a mapping. The first would implement a media object as a timer, which replaces its coordinate system with time. The second would synchronize the media object with a timer, where the coordinate systems of both the object object and the timer are kept intact. Although the first solution seems straightforward, the second solution is preferable for the following reasons:

- It is then possible to share a timer among media objects, so that the timing control (starting, stopping, pausing, fast forwarding and rewinding) can be centralized and shared.

- It provides the flexibility for the user or the client of a media object to decide with which referencing coordinate system to control it. For example, a video object can then be paused at a specified time, or a given frame number.

- As defined on page 290, a *Timer* object is an *ActiveSynchronizable* that has its own process to keep track of time, whereas the media object should have a separate process to "visit" its own coordinates and to present the related content. Implementing a media object as a timer would make it difficult to separate the processes.

The class *TimedSynchronizabe* implements the second solution, applying the Decorator pattern (Gamma et al., 1995; Metsker, 2002), decorating a *Timer* with *ActiveSynchronizable* interfaces. Timed media objects are not immediately modeled, because there is a need for something that is more generic, to schedule tasks that do not have media content to present and that the task execution time is related to a to coordinate system.

To enable event based synchronization between a *TimedSynchronizabe* object and a *Timer*, as already mentioned on page 279, a particular type of time based synchronization events should be defined in order to identify them:

```
┌─ TimedSyncEvent ──────────────────────────
│ SyncEvent
└───────────────────────────────────────────
```

The following *TSEventHandler* couples a *TimedSynchronizable* object with an EventHandler which is only interested in the events from the associated timer:

```
┌─ TSEventHandler ──────────────────────────
│ EventHandler
```

$ts : TimedSynchronizable$

INIT

$iterestedSources = \{ts.timer\}$

and more specifically a *TimedSyncEventHandler*:

TimedSyncEventHandler

$TSEventHandler$

INIT

$interestedEvents = TimedSyncEvent$

$HandleEvent \mathrel{\widehat=} ts.HandleTimedSyncEvent$

which implements the *HandleEvent* operation of an *EventHandler* object (see definition on page 299), forwarding the event, if it is of type *TimedSyncEvetn*, to a associated *TimedSynchronizable* object for further processing.

The *TimedSynchronizable* objects should be controlled, or more precisely, prepared, started, stopped and paused together with the associated *Timer*. Since a *Timer* can be associated with multiple *TimedSynchronizable* objects and the state transitions of the *Timer* should affect all the associated objects, it is convenient to reuse the event handling mechanism to coordinate the states among these objects. Every *TimedSynchronizable* object is then associated with a *StateSyncEventHandler* to handle the state transition events:

StateSyncEventHandler

$TSEventHandler$

INIT

$interestedEvents = TimedSyncEvent$

$HandleEvent \mathrel{\widehat=} ts.HandleStateSyncEvent$

The class *TimedSynchronizable* is defined as follows:

TimedSynchronizable

$\upharpoonright(\cdots, timer)$

$ActiveSynchronizable[Ready_{as}/Ready, Start_{as}/Start,$
$Pause_{as}/Pause, Stop_{as}/Stop]$

$timer : Timer$

$timing : \mathbb{C} \rightarrowtail\!\!\!\!\!\!+\ \mathbb{T}$ [Maps coordinates to time]

$$
\begin{array}{l}
tseHandler : TimedSyncEventHandler \\
sseHandler : StateSyncEventHandler \\
\hline
timing(\!|\ syncSpan\ |\!) = timer.syncSpan \\
repeatNumber = timer.repeatNumber \\
repeatCount = timer.repeatCount \\
direction = timer.direction \\
tseHandler.ts = sseHandler.ts = self
\end{array}
$$

It has a *Timer*, which is made visible to other objects so that the *Timer* object can be controlled directly. The controlling operations inherited from *ActiveSynchrinizable* are renamed and hidden so that later they can be redefined to incorporate the state control of the *timer*. Each *TimedSynchronizable* has an injective partial function *timing* that maps each coordinate in the synchronization span (*syncSpan*) to a different time in the *syncSpan* in the *timer*. Since *timing* is injective, its inverse $timing^{\sim}$ is also a function, by which a given time in the *syncSpan* of the *timer* can also be mapped back to a coordinate in the *syncSpan* of the *TimedSychronizable* object. It is required the *syncSpan* of the *timer* stays as the relational image of the *syncSpac* of the *TimedSychronizable* object under the function *timing*, so that the change on either of them updates another. The state variables *repeatNumber*, *repeatCount* and *direction* are required to be the same as those in the *timer* all the time. The *TimedSynchronizable* object has two event handlers *tseHandler* and *sseHandler* to process timed synchronization events and state transition events respectively.

The initial state of *TimedSynchronizable* object requires the associated objects to be in the initial state as well:

$$
\begin{array}{l}
\textit{I}\small{NIT} \\
\hline
timer.\textit{I}\small{NIT} \land tseHandler.\textit{I}\small{NIT} \land sseHandler.\textit{I}\small{NIT}
\end{array}
$$

The following operation *AddTimedSyncEvents* attaches a *TimedSyncEvent* to each time coordinate of the *timer* if there is not one attached:

$$
\begin{array}{l}
AddTimedSyncEvents \mathrel{\widehat{=}} \bigwedge t : Timer.syncSpan \bullet \\
\quad \textbf{if} \nexists se : timer.syncElements \bullet \\
\qquad\quad \mathrm{first}(se) = t \land \mathrm{second}(se) \in TimedSyncEvent \\
\quad \textbf{then} \left[e : TimedSyncEvent \mid e.source = timer \right] \bullet \\
\qquad\quad timer.attachSyncEvent(t \leadsto t;\ e \leadsto e)
\end{array}
$$

To get the *TimedSynchronizable* object into the ready state, each time in the *syncSpan* of the *timer* is attached with a *TimedSyncEvent*, the event handlers *tseHandler* and *sseHander* are subscribed to the *eventDispatcher* of the *timer*, and then the operation *Ready* of the *timer* is invoked:

$$
\begin{array}{l}
Ready \mathrel{\widehat{=}} AttachTimedSyncEvents \land \\
\qquad timer.eventDispatcher.Subscribe(tseHandler) \land \\
\qquad timer.eventDispatcher.Subscribe(sseHandler) \fatsemi \\
\qquad timer.Ready
\end{array}
$$

The operation *Ready* does not directly put the *TimedSynchronizable* object directly into the ready state. Instead, when the *timer* is ready, it will notify all the event handlers about the state change using its *eventDispatcher*. The *TimedSynchronizable* object will react on the event and finally adapt its state with the *timer*. The same procedure holds for all other state control operations.

The operation *Stop* first unsubscribes its event handlers from the *timer*, then invokes the operation *Stop* of the *timer*:

$$\begin{aligned} Stop \mathrel{\widehat{=}}\ & timer.eventDispatcher.Unsubscribe(tseHandler) \land \\ & timer.eventDispatcher.Unsubscribe(sseHandler) \mathbin{;} \\ & timer.Stop \end{aligned}$$

The operations *Start* and *Pause* are redirected to the corresponding operations in the *timer*:

$$\begin{aligned} & Start \mathrel{\widehat{=}} timer.Start \\ & Pause \mathrel{\widehat{=}} timer.Pause \end{aligned}$$

Whenever the *timer* changes its synchronization state, the state transition event will be dispatched to all interested event handlers and the *TimedSynchronizable* has the following operation to react on the event, changing its own synchronization state accordingly by invoking the controlling operations inherited from its superclass *ActiveSynchronizable*:

$$\begin{aligned} HandleStateSyncEvent \mathrel{\widehat{=}}\ & [e? : StateSyncEvent] \bullet \\ & \textbf{if}\ e?.newState = \mathsf{ready}\ \textbf{then}\ Ready_{as} \\ & \textbf{else if}\ e?.newState = \mathsf{started}\ \textbf{then}\ Start_{as} \\ & \textbf{else if}\ e?.newState = \mathsf{paused}\ \textbf{then}\ Pause_{as} \\ & \textbf{else if}\ e?.newState = \mathsf{stopped}\ \textbf{then}\ Stop_{as} \end{aligned}$$

Timed synchronization events are also received and handled when the *timer* advances its time:

$$\begin{aligned} & HandleTimedSyncEvent \mathrel{\widehat{=}} [e? : TimedSyncEvent] \bullet \\ & \quad \textbf{if}\ timer.current \in \operatorname{ran}(timing) \\ & \qquad \textbf{then}\ curPosition' = timing^{\sim}(timer.current) \mapsto repeatCount \end{aligned}$$

This operation updates the state variable *curPosition* with the position mapped from the *current* time of the *timer*. The *TimedSynchronizable* object has its own active process to step forward, visiting the updated *curPosition*:

$$\begin{aligned} & Step \mathrel{\widehat{=}} \textbf{if}\ syncState = \mathsf{started} \land curPosition = \mathsf{null}\ \textbf{then}\ Stop \\ & \qquad\quad \textbf{else}(Visit \parallel CurToLast) \\ & \textsc{Process} \mathrel{\widehat{=}} \mu\, T \bullet (Step \mathbin{[]} IdleTick) \mathbin{;} T \end{aligned}$$

Note that the operations *Step* and *HandleTimedSyncEvent* are asynchronous. The operation *Step* is carried out in the process of the *TimedSynchronizable* object, but the events are handled in the process of the event dispatcher of the *timer*, or if the event dispatcher is not implemented as an active process, they are handled in the process of the *timer*. In either way, it can be the case that visiting a position takes more time than expected, and *curPosition* is updated more than once. Only the last update will take effect for the next step and the ones in between are skipped. If the *TimedSynchronizable* is a video stream, this may result in frames being dropped.

F.12 *TimedMediaObject*

The class *TimedMediaObject* inherits the active timing behavior from *TimedSynchronizable*, and also as a *MediaObject*, it implements the operations *Ready* and *Retrieve* with a *Prefetcher*:

$$\begin{array}{|l}
\hline
\textit{TimedMediaObject} \\
\upharpoonright(\cdots) \\
\textit{TimedSynchronizable}[\textit{Ready}_{ts}/\textit{Ready}] \\
\textit{MediaObject} \\
\\
\quad \begin{array}{|l}
\hline
\textit{prefetcher} : \textit{Prefetcher} \\
\hline
\textit{prefetcher.positions} = \textit{positions} \\
\textit{prefetcher}.(\_ \prec \_) = (\_ \prec \_) \\
\textit{prefetcher.src} = \textit{src} \\
\hline
\end{array} \\
\\
\textit{Ready} \mathrel{\widehat=} \textit{prefetcher.Prefetch} \;\mathbin{;}\; \textit{Ready}_{ts} \\
\textit{Retrieve} \mathrel{\widehat=} \textit{prefetcher.Reterive} \\
\textit{Visit} = \textit{Visit}_s \;\|\; \\
\qquad (\textit{Retrieve}(\textit{current}) \;\|\; \\
\qquad [\, d? : \mathbb{D}_{\circledcirc} \,] \bullet \textbf{if}\; d? = \mathsf{eof}\; \textbf{then}\; \textit{Stop}\; \textbf{else}\; \textit{Present}) \\
\textit{Step} \mathrel{\widehat=} \textbf{if}\; \textit{syncState} = \mathsf{started} \wedge \textit{curPosition} = \mathsf{null}\; \textbf{then}\; \textit{Stop} \\
\qquad\qquad \textbf{else}\; \textit{Visit} \;\wedge\; \textit{CurToLast} \;\wedge\; \textit{NextVisit} \\
\textsc{Process} \mathrel{\widehat=} \mu\, \textit{AMO} \bullet (\textit{Step} \;[]\; \textit{IdleTick}) \;\mathbin{;}\; \textit{AMO} \\
\hline
\end{array}$$

To ensure immediate *Start* operation, the *TimedMediaObject* must get ready by utilizing a *Prefetcher* object to *Prefetch* certain amount of data. The *Prefetcher* shares the same synchronization positions, the same position order and the same data source as the served *TimedMediaObject* so that the data is prefetched along the same direction as the object steps forward. The *Prefetcher* also guarantees immediate return of data retrieving, though it does not guarantee return with the required data.

An example *Prefetcher* is defined as follows. It implements the prefetching *pool* as a table that holds data with a maximum *capacity*, and indexes the data with the corresponding positions:

Prefetcher

$$
\begin{array}{l}
\mathit{positions} : \mathbb{P}\, \mathit{Position} \\
\_ \prec \_ : \mathit{Position} \leftrightarrow \mathit{Position} \\
\mathit{src} : \mathbb{S} \\
\mathit{pool} : \mathit{Position} \mathrel{+\!\!\!\rightarrow} \mathbb{D} \\
\mathit{readyAmount}, \mathit{capacity} : \mathbb{N} \\
\mathit{next} : \mathit{Position}_{\circledcirc} \\
\mathit{fetch} : \mathbb{C} \rightarrow \mathbb{D}_{\circledcirc} \\
\hline
\mathit{readyAmount} < \mathit{capacity} \\
\#\mathit{pool} < \mathit{capacity} \\
\mathit{next} = \mathbf{let}\ u == \mathit{maximum}(\_ \prec \_, \mathrm{dom}(\mathit{pool})) \bullet \\
\qquad\qquad \mathbf{if}\ u = \mathsf{null} \\
\qquad\qquad \mathbf{then}\ \mathit{minimum}(\_ \prec \_, \mathit{positions}) \\
\qquad\qquad \mathbf{else}\ \mathit{minimum}(\_ \prec \_, \{\mathit{pos} : \mathit{positions} \mid u \prec \mathit{pos}\})
\end{array}
$$

The state variable *readyAmount* defines the least amount of data that is required for immediate *Start* operation and the *capacity* of the *pool* should be big enough to satisfy this requirement. The variable *next* always points to the next position of the maximum element of the *positions* in the *pool*. If there is no such a next position, *next* has a null value. The actual capability of a *MediaObject* to fetch data from the source *src* is modeled as a function *fetch* without any further specification.

When the *Prefetcher* is invoked to *Retrieve* data related to a given coordinate, it serves the data directly from the *pool*. However if the required data is not yet in the pool, a null value is returned. It might be the case that the *pool* is so big that multiple positions in the pool have the same coordinate, then the first one in the order of "$\prec$" is served. Once a position is served from the pool, the positions prior to this position are removed from the pool to leave the space for pooling the data from new positions:

Retrieve

$$
\begin{array}{l}
\Delta(\mathit{pool}) \\
c? : \mathbb{C} \\
d! : \mathbb{D}_{\circledcirc} \\
p : \mathit{Position}_{\circledcirc} \\
\hline
p = \mathit{minimum}(\_ \prec \_, \{p : \mathrm{dom}(\mathit{pool}) \mid \mathrm{first}(p) = c?\}) \\
\mathbf{if}\ p \neq \mathsf{null}\ \mathbf{then}\ d! = \mathit{pool}(p) \wedge \\
\qquad\qquad\qquad \mathit{pool}' = \{\mathit{pos} : \mathit{positions} \mid p \prec \mathit{pos}\} \lhd \mathit{pool} \\
\mathbf{else}\ d! = \mathsf{null}
\end{array}
$$

The operation *Pooling* first clears the data that are not related to the current *positions* from the *pool*, then if the *pool* is not full, fetches the data of the *next* position into the *pool*:

$$\begin{aligned}
&\mathit{Pooling} \mathrel{\widehat=} \mathit{pool} := (\mathit{positions} \lhd \mathit{pool}) \mathbin{⨾} \\
&\qquad [\#\mathit{pool} < \mathit{capacity} \land \mathit{next} \neq \mathsf{null}] \land \\
&\qquad \mathbf{let}\ n == \mathit{fetch}(\mathrm{first}(\mathit{next})) \bullet \\
&\qquad \mathbf{if}\ n \neq \mathsf{null}\ \mathbf{then}\ \mathit{pool} := \mathit{pool} \cup \mathit{next} \mapsto n
\end{aligned}$$

The operation *Prefetch* recursively keeps pooling the data until the amount of the data in the *pool* has reached the "*readyAmount*":

$$\begin{aligned}
&\mathit{Prefetch} \mathrel{\widehat=} \mathbf{if}\ \#\mathit{pool} < \mathit{readyAmount} \land \#\mathit{pool} < \#\mathit{positions} \\
&\qquad \mathbf{then}\ \mathit{Pooling} \mathbin{⨾} \mathit{Prefetch}
\end{aligned}$$

While the *TimedMediaObject* is busy presenting the data retrieved from the *pool*, the *Prefetcher* should run as a separate process that keeps pooling the data so that continuous presentation is possible:

$$\begin{aligned}
&\mathit{IdleTick} \mathrel{\widehat=} [\neg\ \mathrm{pre}(\mathit{Pooling}) \land \tau' = \tau + 1] \\
&\mathrm{PROCESS} \mathrel{\widehat=} \mu\, F \bullet \mathit{Pooling} \mathbin{[\!]} \mathit{IdleTick} \mathbin{⨾} F
\end{aligned}$$

F.13 *EventDispatcher*

An *EventDispatcher* is necessary for every *Synchronizable* object to send synchronization events around to those which are interested in certain types of events. The Reactor pattern (Schmidt et al., 2000) is employed:

EventDispatcher

$$\upharpoonright(\mathit{Subscribe}, \mathit{Unsubscribe}, \mathit{Dispatch})$$

$$\mathit{handlers} : \mathbb{P} \downarrow \mathit{EventHandler}$$

INIT

$$\mathit{handlers} = \varnothing$$

$$\begin{aligned}
&\mathit{Subscribe} \mathrel{\widehat=} [\mathit{handler}? : \downarrow \mathit{EventHandler}] \bullet \\
&\qquad \mathit{handlers} := (\mathit{handlers} \cup \{\mathit{handler}?\}) \\
&\mathit{Unsubscribe} \mathrel{\widehat=} [\mathit{handler}? : \downarrow \mathit{EventHandler}] \bullet \\
&\qquad \mathit{handlers} := (\mathit{handlers} \setminus \{\mathit{handler}?\}) \\
&\mathit{Dispatch} \mathrel{\widehat=} [e? : \downarrow \mathit{Event}] \bullet \\
&\qquad \textstyle\bigwedge h : \mathit{handlers} \mid e? \in h.\mathit{interestedEvents} \land \\
&\qquad\qquad e?.\mathit{source} \in h.\mathit{interestedSources} \bullet \\
&\qquad\qquad h.\mathit{HandleEvent}(e?)
\end{aligned}$$

where the *EventDispatcher* keeps a *handlers* registry of the target *EventHandler* objects through the operations Subscribe and Unsubscribe, dispatches the events to these handlers if the events and the sources are of their interests.

The Reactor pattern is related to the Observer pattern, where all the dependents are informed when a single subject changes. In the Reactor pattern, a single handler is informed when an event of interest to the handler occurs on a source of events (Schmidt et al., 2000). The observer pattern is not enough in this situation, because once the event is triggered, neither the *Synchronizable* nor the dispatcher should store the dispatched events for the observers to check what has happened – events are not part of the observable state. Further, the event handlers, which are receiving the events, should only be associated with particular types and sources of Events – the Observer pattern is not designed for this purpose.

The Proactor (Schmidt et al., 2000) can also be used to support *asynchronously* demultiplexing and dispatching of multiple events. The *ActiveEventDispatcher* keeps the incoming events in a queue and dispatches the events from the queue in a separate process:

ActiveEventDispatcher

$\upharpoonright(Subscribe, Unsubscribe, Dispatch)$

$EventDispatcher[Dispatch_{ed} / Dispatch]$

$events : \mathrm{seq} \downarrow Event$

INIT

$events = \langle\, \rangle$

$Enqueue \mathrel{\widehat=} [\, e? : \downarrow Event \,] \bullet events := (events \frown \langle e? \rangle)$

$Dequeue \mathrel{\widehat=} [\, \Delta(events) e! : \downarrow Event \mid events = \langle e! \rangle \frown events' \,]$

$Dispatch \mathrel{\widehat=} Enqueue$

$Dispatching \mathrel{\widehat=} Dequeue \fatsemi Dispatch_{ed}$

$IdleTick \mathrel{\widehat=} [\, \neg\ \mathrm{pre}(Dispatching) \wedge \tau' = \tau + 1 \,]$

$\mathrm{PROCESS} \mathrel{\widehat=} \mu\, AED \bullet Dispatching \mathbin{[]} IdleTick \fatsemi AED$

F.14 *EventHandler*

The *EventHandler* contains the information about interested events and originating sources. It implements a single operation, *HandleEvent*, which is used by the *EventDispatcher* to dispatch events. The event itself, carries the information of its type, and the source of the event:

EventHandler

$interestedEvents : \mathbb{P} \downarrow Event$

$interestedSources : \mathbb{P}\, \mathbb{O}$

$HandleEvent \mathrel{\widehat=} [\, e? : \downarrow Event \,]$

The advantage of the single-method interface is that it is possible to add new types of events without changing the interface and existing event handlers. However this approach encourages the use of switch statements in the subclass's *HandleEvent* method. This can be overcome by partitioning the events, and coupling a particular type of *EventHandler* to the actual handling objects as it is demonstrated in the *TimedMediaObject* (see specification on page 292). An alternative solution is to define separate virtual hook operations for each type of the interested events in the

EventHandler so that it is easier to selectively override operations in the base class and avoid further demultiplexing (Schmidt et al., 2000). However it requires the types of the events to be set in advance, which is rather not possible in designing synchronizable objects at an abstract level.

BACKGROUND MATERIAL G

Specifications of the Action Service Factory Pattern

This background material presents the formal specification of the components of the Action Service Factory that is introduced in section 6.3 of chapter 6.

G.1 *SyncService* and concrete actions

The Timed Action pattern requires the operations of the *ActionService* to have unified input and output interfaces

$$[p?, r! : \mathit{Dictionary}]$$

where the input parameters and the output results are wrapped into $\mathit{name} \mapsto \mathit{value}$ pairs. To apply this pattern to *Synchronizable* objects so that the synchronization operation can be provided as action services, these operations must be adapted to yield to the $\mathit{name} \mapsto \mathit{value}$ interface, using the Adapter pattern (Gamma et al., 1995; Metsker, 2002).

The class *SyncService* extends the *ActionService*, renaming $Op_i\ (i : \mathit{ActionID})$ to real operation names. A *SyncService* is an "Adapter" for a synchronizable object that is referred as *sbl*:

SyncService

$$\upharpoonright(\mathit{Ready}, \mathit{Start}, \mathit{SetBegin}, \mathit{GetBegin}, \cdots)$$
$$\mathit{ActionService}[\mathit{Ready}/Op_1, \mathit{Start}/Op_2, \mathit{SetBegin}/Op_3, \mathit{GetBegin}/Op_4, \cdots]$$

$$\boxed{\mathit{sbl} : \downarrow \mathit{Synchronizable}}$$

The control operations such as *Ready* and *Start* from the *Synchronizable* object *sbl* do not have any input and output. They are wrapped with the operation $[p? :$

Dictionary$]$ that accepts any input, and the operation $[\,r! : \mathit{Dictionary} \mid r! = \varnothing\,]$ that returns an empty set of $\mathit{name} \mapsto \mathit{value}$ pairs. Any input is accepted, but will be silently ignored:

$$
\begin{array}{|l}
\mathit{Ready} \mathrel{\widehat=} [\,p? : \mathit{Dictionary}\,] \mathbin{\S} \mathit{sbl.Ready} \mathbin{\S} [\,r! : \mathit{Dictionary} \mid r! = \varnothing\,] \\
\mathit{Start} \mathrel{\widehat=} [\,p? : \mathit{Dictionary}\,] \mathbin{\S} \mathit{sbl.Start} \mathbin{\S} [\,r! : \mathit{Dictionary} \mid r! = \varnothing\,] \\
\ldots
\end{array}
$$

For those operations from *sbl* that have inputs and outputs, the input parameters in form of $\mathit{name} \mapsto \mathit{value}$ pairs are passed through a "decoding" operation that outputs the parameters for the operation from *sbl*. After the operation from *sbl* is invoked, the outputs from this operation is then put into another operation that "encodes" the results in form of $\mathit{name} \mapsto \mathit{value}$ pairs again. An example is how to adapt *sbl.SetBegin* to the required input and output interface:

$$
\begin{array}{|l}
\mathit{SetBegin} \mathrel{\widehat=} [\,p? : \mathit{Dictionary};\ \mathit{begin}! : \mathbb{C}_{\circledcirc} \mid \mathit{begin}! = \mathit{Get}(p?, \text{“}\mathit{begin}\text{”})\,] \mathbin{\S} \\
\qquad\qquad \mathit{sbl.SetBegin} \mathbin{\S} [\,r! : \mathit{Dictionary} \mid r! = \varnothing\,] \\
\ldots
\end{array}
$$

In addition, a *SyncService* object may also provide "getter" operations for visitable state variables of the object *sbl* so that state queries to *sbl* can be scheduled too:

$$
\begin{array}{|l}
\mathit{GetBegin} \mathrel{\widehat=} [\,p?, r! : \mathit{Dictionary} \mid r! = \{\text{“}\mathit{begin}\text{”} \mapsto \mathcal{O}(\mathit{sbl.begin})\}\,] \\
\ldots \\
\hline
\end{array}
$$

All these operations that will be available to the clients are then encapsulated as corresponding *Action* objects so that they can be stored and retrieved for scheduling. For example, the operations *Ready* and *Start* are encapsulated in

$$
\begin{array}{|l}
\hline
\mathit{ReadyAction} \\
\mathit{Action} \\
\mathit{Execute} \mathrel{\widehat=} \mathit{as.Ready}(p) \mathbin{\S} \mathit{tr.Complete} \\
\hline
\end{array}
$$

and

$$
\begin{array}{|l}
\hline
\mathit{StartAction} \\
\mathit{Action} \\
\mathit{Execute} \mathrel{\widehat=} \mathit{as.Start}(p) \mathbin{\S} \mathit{tr.Complete} \\
\hline
\end{array}
$$

respectively.

G.2 *SyncServiceProxy*

The class *SyncServieProxy* implements the interfacing class *ActionServiceProxy* with the concreted actions defined for the class *SyncService*. The interfaces are created for the client to schedule the synchronization operations with an extra time parameter:

SyncServiceProxy

$\upharpoonright((\textsc{Init}, as, eq, s), Ready, Start, \ldots)$

$ActionServiceProxy[SyncActionService/ActionService,$
$\quad Ready/Op_1, Start/Op_2, \cdots]$

NewTimedReadyAction

$t? : \mathbb{T}_{\circledcirc}$
$p? : Dictionary$
$ta! : TimedAction$
$tr! : TentativeResult$

$\text{new}(ta!, TimedAction) \land \text{new}(tr!, TentativeResult)$
$\text{new}(ta!.a, ReadyAction) \land ta!.a.\textsc{Init}$
$ta!.a.as = as \land ta!.a.eq = eq \land ta!.a.p = p? \land ta!.a.tr = tr!$
$\textbf{if}\ t? = \mathsf{null}\ \textbf{then}\ ta!.t = \tau\ \textbf{else}\ ta!.t = t?$

NewTimedStartAction

$\ldots$

$\ldots$

$Ready \mathrel{\widehat{=}} NewTimedReadyAction[o!/ta!] \mathbin{⨟} s.Subscribe$
$Start \mathrel{\widehat{=}} NewTimedStartAction[o!/ta!] \mathbin{⨟} s.Subscribe$
$\ldots$

So far the *SyncService* for the synchronization tasks have been defined on a *Synchronizable* object , the concrete *Actions* that wrap these tasks as objects and the *SyncServiceProxy* interfaces for the client to schedule these tasks. The other participants in the Timed Action pattern, such as *TimedAction*, *ExecutionQueue*, *TentativeResult* and *Scheduler* (see figure 6.1 on page 68), can be employed directly without changing the definition.

The next could be defining the factories that dynamically create all needed participants in the Timed Action pattern and link them all together for a given media object according to its type. However it is necessary to understand and express these media types first.

G.3 Media types

Let's introduce a type to identify different types of media content

$[\mathbb{M}]$

and a global relation $\rhd$ on this type

$\_ \rhd \_ : \mathbb{M} \leftrightarrow \mathbb{M}$

$\forall m : \mathbb{M} \bullet \neg\,(m \rhd m)$ [Irreflective]
$\forall m_1, m_2, m_3 : \mathbb{M} \bullet m_1 \rhd m_2 \land m_2 \rhd m_3 \Rightarrow m_1 \rhd m_3$ [Transitive]
$\forall m_1, m_2 : \mathbb{M} \bullet m_1 \rhd m_2 \Rightarrow \neg\,(m_2 \rhd m_1)$ [Antisymmetric]

that evaluates whether a given media type "is a" subtype of another media type, for example, $mp3$ is a subtype *audio* type, $mp4$ is a subtype of *video*, and *audio* and *video* are subtypes of *timedmedia*. For the convenience, its reflective version $\trianglerighteq$ may also be used. Note that $\triangleright$ and $\trianglerighteq$ are partial relations and one shall not model these different media types into a static class hierarchy. Such a static hierarchy is feasible only when the entire inheritance tree is given before the system is up and running. It is necessary to deal with a dynamic media ontology that is defined by both the content itself and playback components in the runtime. The issue of media compatibility is a complex topic in itself. Here a pragmatic approach is adopted.

The following global total function finds one of the *closest* type of a given media type from a set of media types:

$$\begin{array}{l} closest == \lambda\, m : \mathbb{M};\ M : \mathbb{P}\,\mathbb{M} \bullet \\ \qquad\qquad \mathbf{let}\, a == \{m_1 : M \mid m \trianglerighteq m_1 \wedge (\nexists m_2 : M \bullet m \triangleright m_2 \triangleright m_1)\} \bullet \\ \qquad\qquad\quad (m_0 : M_{\circledcirc} \mid \mathbf{if}\, a = \varnothing\ \mathbf{then}\, m_0 = \mathsf{null}\ \mathbf{else}\, m_0 \in a) \end{array}$$

If the given type m is in the set M, the *closest* one is itself, otherwise the function tries to find a type in M such that there is no other types in between in chain of $\triangleright$. If the type is not found in M, a null value is returned to indicate the situation.

Let's also define a global total function that determines the media type from a given source:

$$\mid\ stype : \mathbb{S} \rightarrow \mathbb{M}_{\circledcirc}$$

which also includes the case that the media type can not be determined. In this case, the function *stype* returns a null value.

G.4 *AbstractSyncFactory*

The class *AbstractSyncFactory* defines the interfaces for creating synchronizable objects for the given media sources, and creating event dispatchers for these synchronizable objects:

$$\begin{array}{|l} \textit{AbstractSyncFactory} \\ \hline \upharpoonright(CreateSyncronizable) \\ CreateEventDispatcher \mathrel{\widehat=} [\, ed! : \downarrow EventDispatcher \,] \\ CreateSynchronizable_0 \mathrel{\widehat=} [\, src? : \mathbb{S};\ sbl! : \downarrow Synchronizable_{\circledcirc} \,] \\ AttachEventDispatcher \mathrel{\widehat=} \\ \qquad [\, sbl? : \downarrow Synchronizable_{\circledcirc};\ ed? : EventDispatcher \,] \bullet \\ \qquad \mathbf{if}\, sbl? \neq \mathsf{null}\ \mathbf{then}\, sbl?.SetEventDispatcher(ed?) \\ CreateSynchronizable \mathrel{\widehat=} ((CreateEventDispatcher \parallel \\ \qquad\qquad\qquad\qquad\qquad CreateSynchronizable_0) \parallel_! \\ \qquad\qquad\qquad\qquad\qquad AttachEventDispatcher) \setminus (ed!) \\ \hline \end{array}$$

The parallel composition operator $\parallel_!$ is a variation of the parallel operator $\parallel$. The difference is that $\parallel_!$ does not hide the output of a matching input/output communication pair. The operation *CreateSynchronizable* hides the output *ed*! but leaves the created *sbl*! exposed.

It is up to the concrete factories to decide which types synchronizable objets should be created and how, and to decide which event dispatching strategy (active or passive, shared or multiple) should be applied. It also suggests the inheriting factories should only make the operation *CreateSynchronizable* visible to the clients.

G.5 SyncFactoryImpl

Let's give an example definition of the class *SyncFactoryImpl*, which creates synchronizable objects according to its capability. The capability is modeled as a partial function *cap* that maps a given media type to a set of synchronizable objects.

$$
\begin{array}{l}
\textit{SyncFactoryImpl} \\
\upharpoonright(\textit{CreateSyncronizable}) \\
\textit{AbstractSyncFactory} \\
\hline
\quad cap : \mathbb{M} \nrightarrow \mathbb{P}\downarrow\textit{Synchronizable} \\
\quad created : \mathbb{P}\downarrow\textit{Synchronizable} \\
\quad ed : \downarrow\textit{EventDispatcher}_{\circledcirc}
\end{array}
$$

It also keeps track of the created synchronizable objects. A created object should not be created for a second time until recycled. This example factory maintains an *EventDispatcher* variable *ed* for all created objects. Upon initiation, *ed* is set to null and *created* is empty:

$$
\begin{array}{l}
\textsc{Init} \\
\hline
ed = \mathsf{null} \wedge created = \varnothing
\end{array}
$$

The following operation creates a new *ActiveEventDispatcher* object as when being invoked for the first time, and outputs this dispatcher ever since:

$$
\begin{array}{l}
\textit{CreateEventDispatcher} \\
\Delta(ed) \\
ed! : \downarrow\textit{EventDispatcher} \\
\hline
\textbf{if}\ ed = \mathsf{null}\ \textbf{then}\ \mathrm{new}(ed', \textit{ActiveEventDispatcher}) \wedge ed! = ed' \\
\qquad \textbf{else}\ ed! = ed
\end{array}
$$

The following operation outputs a synchronizable object according to the media type of the source from the input. It first checks whether the source is of a valid media type then finds the "closest" media type in its capability list. If it is capable of dealing with this type of media, a synchronizable object, if still available, will be delivered to the client. The delivered object is also added to the set of "created" so that it won't be delivered again.

$$
\begin{array}{|l}
\textit{CreateSynchronizalbe}_0 \\
\hline
\Delta(\textit{created}) \\
\textit{src}? : \mathbb{S} \\
\textit{sbl}! : \downarrow \textit{Synchronizable}_{\circledcirc} \\
\hline
\textbf{let}\ m == \textit{stype}(\textit{src}?) \bullet \\
\quad \textbf{if}\ m = \mathsf{null}\ \textbf{then}\ \textit{sbl}! = \mathsf{null} \\
\qquad \textbf{else let}\ \textit{found} == \textit{closest}(m, \operatorname{dom}(\textit{cap})) \bullet \\
\qquad\quad \textbf{if}\ \textit{found} = \mathsf{null}\ \textbf{then}\ \textit{sbl}! = \mathsf{null} \\
\qquad\qquad \textbf{else let}\ \textit{available} == \textit{cap}(\textit{found})) \setminus \textit{created} \bullet \\
\qquad\qquad\quad \textbf{if}\ \textit{available} = \varnothing\ \textbf{then}\ \textit{sbl}! = \mathsf{null} \\
\qquad\qquad\qquad \textbf{else}\ \textit{sbl}! \in \textit{available} \wedge \textit{created}' = \textit{created} \cup \{\textit{sbl}!\} \\
\textit{sbl}! \in \textit{MediaObject} \Rightarrow \textit{sbl}!.\textit{src} = \textit{src}?
\end{array}
$$

The sets of *Synchronizable* objects in the range of the function *cap* can be both infinite and finite, however in practice, the factory might only be capable of creating a certain number of a particular type of *Synchronizable* objects because of for example limited processing or presentation resources. So often in implementation, it is important to recycle the no longer used *Synchronizable* objects so that they can be delivered again to the clients:

$$
\begin{array}{|l}
\textit{RecycleSynchronizable} \mathrel{\widehat{=}} [\, \textit{sbl}? : \downarrow \textit{Synchronizable} \,] \bullet \\
\qquad\qquad\qquad\qquad\qquad \textit{created} := (\textit{created} \setminus \{\textit{sbl}?\}) \\
\hline
\end{array}
$$

G.6 AbstractSyncServiceFactory

Once a *Sychronizable* object is created, the next step is to wrap it in a *SyncService* object and create a *SyncServiceProxy* so that the client can schedule the synchronization tasks. The class *AbstractSyncServiceFactory* encapsulates these object creation processes and provides a single interface *CreateSyncServiceProxy* that takes a *Synchronizable* object as input and delivers the related *SyncServiceProxy* to the client. It is still abstract, because it only defines the output interface for the operation *CreateScheduler* and leaves the details to the concrete factory. The concrete factory may then decide whether every proxy should have a separate scheduling process, or share one.

$$
\begin{array}{|l}
\textit{AbstractSyncServiceFactory} \\
\hline
\upharpoonright(\textit{CreateSyncServiceProxy}) \\
\textit{CreateScheduler} \mathrel{\widehat{=}} [\, s! : \textit{Scheduler} \,] \\
\textit{CreateSyncService} \mathrel{\widehat{=}} [\, \textit{sbl}? : \downarrow \textit{Synchronizable};\ \textit{ss}! : \textit{SyncService} \mid \\
\qquad\qquad\qquad\qquad \textit{ss}!.\textit{sbl} = \textit{sbl}! \,] \\
\textit{CreateSyncServiceProxy} \mathrel{\widehat{=}} \textit{CreateScheduler} \parallel \textit{CreateSyncService} \parallel \\
\qquad [\, s? : \textit{Scheduler};\ \textit{ss}? : \textit{SyncService};\ \textit{ssp}! : \textit{SyncServiceProxy}_{\circledcirc} \mid \\
\qquad\ \textbf{if}\ \textit{ss}? = \mathsf{null}\ \textbf{then}\ \textit{ssp}! = \mathsf{null}\ \textbf{else}\ \textit{ssp}!.\textit{as} = \textit{ss}? \wedge \textit{ssq}!.s = s? \,] \\
\hline
\end{array}
$$

G.7 SyncServiceFactoryImpl

The following is an example implementation of the class *SyncServiceFactoryImpl*. It implements the operation *CreateScheduler* from its abstract super class, by creating a *Scheduler* object when it is invoked for the first time, and returns this object ever after.

```
SyncServiceFactoryImpl
    $\upharpoonright(CreateSyncServiceProxy)$
    $AbstractSyncServiceFactory$

    $s : Scheduler_{\circledcirc}$

    INIT
        $s = \mathsf{null}$

    CreateScheduler
        $\Delta(s)$
        $s! : Scheduler$
        ----
        $\textbf{if}\ s = \mathsf{null}\ \textbf{then}\ \mathrm{new}(s', Scheduler) \wedge s' = s!\ \textbf{else}\ s! = s$
```

G.8 An Example

To provide an overview of how everything comes together, let's first assume there is an actor who owns a *syncFactory* of the type *SyncFactoryImpl* and a *syncServiceFactory* of the type *SyncServiceFactoryImpl*:

$$syncFactory : SyncFactoryImpl;$$
$$syncServiceFactory : SyncServiceFactoryImpl.$$

The syncFactory has a list of capacities:

$$syncFactory.cap = \{mp3 \mapsto Mp3MediaObject, audio \mapsto AudioMediaObject\}$$

which claims that it can create not only dedicated $mp3$ media objects that are specially designed for playing back $mp3$'s, but also more generic *audio* media objects that play back many media types:

$$mp3 \rhd audio \wedge wav \rhd audio \wedge au \rhd audio.$$

Both *Mp3MediaObject* and *AudioMediaObject* extend *TimedMediaObject* to use a *Timer* and a *Prefetcher* to present streamed audio.

Suppose the actor receives a request from the client to play a live $mp3$ stream at a specific time t_0, and when that is ended, play another piece of *wav* file but the first 2 seconds of the *wav* are requested to be skipped. Assume the $mp3$ is from the source "$http : //www.server.com/file.mp3$" $\in \mathbb{S}$:

$$stype(\text{“}http : //www.server.com/file.mp3\text{”}) = mp3$$

and the *wav* is from the local storage "*file* : ///*c* : /*mymusic*/*file.wav*" $\in \mathbb{S}$:

$$stype(\text{“}file : ///c : /mymusic/file.wav\text{”}) = wav.$$

The actor uses the source of *mp*3 as input to the *syncFactory* to get a synchronizable object

$$syncFactory.CreateSynchronizable(src \rightsquigarrow \text{“}http : //www.server.com/file.mp3\text{”}).$$

Since

$$closest(mp3, \mathrm{dom}(syncFactory.cap) = mp3,$$

checking against the capability list *syncFactory.cap*, *syncFactory* creates a *MP*3*MediaObject* as output. Let's denote it as *mp*3*object*. Using a sequential composition, this output can then be used as the input for *syncServiceFactory* to wrap it with an action proxy:

$$syncServiceFactory.CreateSyncServiceProxy(sbl \rightsquigarrow mp3object).$$

Let's write *mp*3*proxy* to denote the *SyncServiceProxy* product of this operation, and *mp*3*service* to denote the intermediate *SyncService* product. Note that

$$mp3proxy.as = mp3service \wedge mp3service.sbl = mp3object.$$

The actor uses the same procedure to create the *Synchronizable* object and the proxy for the action service. Although there is no dedicated media object for *wav*, the *syncFactory* finds that the *AudioMediaObject* can deal with *audio*, and *wav* is an immediate subtype of *audio*. A *AudioMediaObject* is produced instead. Let's write *wavobject* to denote this product. The *wavobject* is then sent to *syncServiceFactory* to get a product of *SyncServiceProxy*. Let's refer this product as *wavproxy*. The intermediate *SyncService* product will be referred as *wavservice*, where

$$wavproxy.as = wavservice \wedge waveservice.sbl = wavobject.$$

Notice that the *syncFactory* has assigned the same *ActiveEventDispatcher* object for both *mp*3*object* and *wavobject*, and the *syncServiceFactory* has also assigned the same *Scheduler* object for both *mp*3*proxy* and *wavproxy*. That is,

$$mp3object.eventDispatcher = wavobject.eventDispatcher \wedge \\ mp3proxy.s = wavproxy.s.$$

The *mp*3*object* has to get ready before the time t_0:

$$mp3object.Ready \ \wedge \ \left[\tau' < t_0\right].$$

The *wavobject* should skip the first 2 seconds of the content. Let's assume that 1 second takes 1000 time units in the system, then the synchronization positions of the *wavobject* should be set by invoking:

$$wavobject.SetBegin(2000)$$

As requested, the presentation of the *wavobject* should start when the *mp3object* stops. However it is not known in advance when the *mp*3 stream is going to be ended, and it is possible that the stream is not alive and the *mp3object* can be immediately stopped. So the *wav* object also has to get ready before the time t_0:

$$wavobject.Ready \;\wedge\; \left[\tau' < t_0\right].$$

Both the *mp3object* and the *wavobject* will then initiate their *Timer* objects, activate their *Prefetcher* objects to *Retrieve* enough amount of data so that they are ready for immediate and continuous presentation.

To detect the event of *mp3object* stopping, the actor has to implement an *eventHandler* of the type *EventHandler*, where

$$eventHandler.interestedEvents = \{e : StateSyncEvent \mid e.newState = \mathsf{stopped}\}$$
$$eventHandler.interestedSources = \{mp3object\}$$

and the operation *eventHandler.HandleEvent* should be managed by the actor to invoke *wavobject.Start*. The *eventHandler* is registered to the event dispatcher:

$$mp3object.eventDispatcher.Subscribe(eventHandler)$$

Now the actor is ready to schedule the presentation by accepting a request through the action proxies. For example:

$$mp3proxy.Start(t \leadsto t_0, p \leadsto \varnothing) \setminus (tr!)$$

which will create a *TimedAction* object, denoted as *ta*, which wraps up an *StartAction* object, denoted as *sa*, such that

$$ta.t = t_0 \wedge ta.a = sa$$

and

$$sa.as = mp3servcie \wedge sa.eq = mp3proxy.eq \wedge sa.p = \varnothing.$$

In this example, the actor does not care about the output *TentativeResult* object *tr*! and it is then hidden from the environment. The created *TimedAction* object *ta* is subscribed to the Scheduler *mp3proxy.s*, and obviously now it can be asserted that

$$ta \in mp3proxy.s.tas.$$

The scheduler *mp3proxy.s* is an active object (see definition on page 271) and it keeps checking its subscriber list *tas* while time goes on. When the scheduled time $ta.t = t_0$ has reached, the operation *ta.Update*(*self*) is invoked. This will result in the wrapped *StartAction* object *sa* to be enqueued in *sa.eq* (see definition on page 270), that is, *mp3proxy.eq*, which is an active object of the type *ExecutionQueue* (see definition on page 268). So far nothing has been enqueued except *sa*. The active Process immediately dequeues *sa* for execution, which in turn results in the operation *sa.Execute* being invoked, that is, $sa.as.Start(p) \,⨾\, sa.tr.Complete$. Since $sa.as =$

mp3service, this will finally call the operation *mp3object.Start* through *mp3service.Start*: $[p? : Dictionary] \fatsemi sbl.Start \fatsemi [r! : Dictionary \mid r! = \varnothing]$ in which it is known that $mp3service.sbl = mp3object$. The input, $p? = sa.p = \varnothing$, is ignored.

Remember the *mp3object* has already been ready for presentation. The operation *mp3object.Start* will immediately activate the *Timer* object, which will then actively push the *mp3object* forward by mapping the time to data positions, retrieving the data from the pool of the *Prefetcher*, and presenting it to the real world (see the definition of *TimedMediaObject* on page 296).

The *mp3object* is now started, presenting its beats and rhymes, until an eof is received from the source, or for whatever reasons its *Stop* operation is invoked. The *mp3object*'s state is then changed to stopped.

This state transition will issue a *SyncStateEvent* which *newState* is stopped and which *source* is the *mp3object*. The event dispatcher, *mp3obect.eventDispatcher*, picks this event up and distributes it to the interested event handlers (see specification on page 282).

There was a *eventHandler* subscribed to this type of event. The operation *eventHandler.handleEvent* is called by the dispatcher and hence the operation *wavobject.Start*. Now, immediately after the *mp3object* is stopped, the *wavobject* starts its presentation without any delay, because it has already been ready and waiting for this starting trigger.

BACKGROUND MATERIAL H

Specifications of the Channel Pattern

This background material presents the formal specification of the components of the Channel pattern that is introduced in section 7.2 of chapter 7.

H.1 *Channel*

The class *Channel* provides administrative operations through two interfacing objects:

- The *ConsumerAdmin* interface allows consumers to be connected to the channel, and the operation *ForConsumers* returns an object reference that supports the *ConsumerAdmin* interface.
- The *SupplierAdmin* interface allows suppliers to be connected to the channel, and the operation *ForSuppliers* returns an object reference that supports the *SupplierAdmin* interface.

Channel

$$\upharpoonright(\mathit{ForSuppliers}, \mathit{ForConsumers}, \mathit{Destroy}, (\mathit{Transfer}))$$

$$\begin{array}{|l}
\mathit{supplierAdmin} : \downarrow \mathit{SupplierAdmin}_{©} \\
\mathit{consumerAdmin} : \downarrow \mathit{ConsumerAdmin}_{©} \\
\hline
\mathit{consumerAdmin.channel} = \mathit{supplierAdmin.channel} = \mathit{self}
\end{array}$$

$$\mathit{ForSuppliers} \mathrel{\widehat{=}} [\,\mathit{supplierAdmin}! : \{\mathit{supplierAdmin}\}\,]$$
$$\mathit{ForConsumers} \mathrel{\widehat{=}} [\,\mathit{consumerAdmin}! : \{\mathit{consumerAdmin}\}\,]$$

The *Destroy* operation destroys the channel and both *ConsumerAdmin* and *SupplierAdmin* objects:

$$\mathit{Destroy} \mathrel{\widehat{=}} \mathit{supplierAdmin.Destroy} \parallel \mathit{consumerAdmin.Destroy}$$

The protected operation *Transfer* transmits the data from the *supplierAdmin* to *consumerAdmin*. How this should be done depends on the QoS requirements and data dispatching strategy. A minimum implementation can be described here: it pushes the input data from the *SupplierAdmin* object directly to the *ConsumerAdmin* object:

$$Transfer \mathrel{\widehat{=}} [\, d? : \mathbb{O} \,] \bullet consumerAdmin.PushFromChannel(d?)$$

All data to be transferred are objects of any type. If the data to be transferred is not an object, it easy to wrap any non-object data up as an object (see specification on page 266).

H.2 *SupplierAdmin*

The class *SupplierAdmin* creates the proxy consumers for suppliers and keeps track of these connected proxy consumers. It defines the first step for connecting suppliers to the channel (see the second step later defined in classes *ProxyPushConsumer* on page 319 and *ProxyPullConsumer* on page 320); clients use it to obtain proxy consumers:

SupplierAdmin

$\upharpoonright(ObtainPushConsumer, ObtainPullConsumer,$
$\quad DisconnectPushConsumer, DisconnectPullConsumer,$
$\quad (Destroy, PushToChannel))$

$channel : \downarrow Channel$
Δ
$pushConsumers : \mathbb{P} \downarrow ProxyPushConsumer_{©}$
$pullConsumers : \mathbb{P} \downarrow ProxyPullConsumer_{©}$

The operation *ObtainPushConsumer* creates and outputs a *ProxyPushConsumer* object. The *ProxyPushConsumer* object can then be used to connect a push-style supplier. The operation *DisconnectPushConsumer* removes the *ProxyPushConsumer* object from *pushConsumers*

ObtainPushConsumer

$pushConsumer! : \downarrow ProxyPushConsumer$

$\mathrm{new}(pushConsumer!, ProxyPushConsumer)$
$pushConsumer!.\textsc{Init}$
$pushConsumer!.supplierAdmin = self$
$pushConsumers' = pushConsumers \cup \{pushConsumer!\}$

$$\begin{array}{|l}
\textit{DisconnectPushConsumer} \\
\hline
\textit{pushConsumer}? : \downarrow \textit{ProxyPushConsumer} \\
\hline
\textit{pushConsumers}' = \textit{pushConsumers} \setminus \{\textit{pushConsumer}?\} \\
\hline
\end{array}$$

The operation *ObtainPullConsumer* creates and outputs a *ProxyPullConsumer* object. The *ProxyPullConsumer* object can then be used to connect a pull-style supplier. The operation *DisconnectPullConsumer* removes the *ProxyPullConsumer* object from *pullConsumers*:

$$\begin{array}{|l}
\textit{ObtainPullConsumer} \\
\hline
\textit{pullConsumer}! : \downarrow \textit{ProxyPullConsumer} \\
\hline
\mathrm{new}(\textit{pullConsumer}!, \textit{ProxyPullConsumer}) \\
\textit{pullConsumer}!.\textsc{Init} \\
\textit{pullConsumer}!.\textit{supplierAdmin} = \textit{self} \\
\textit{pullConsumers}' = \textit{pullConsumers} \cup \{\textit{pullConsumer}!\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{DisconnectPullConsumer} \\
\hline
\textit{pullConsumer}? : \downarrow \textit{ProxyPullConsumer} \\
\hline
\textit{pullConsumers}' = \textit{pullConsumers} \setminus \{\textit{pullConsumer}?\} \\
\hline
\end{array}$$

The protected operation *Destroy* is called by the *Destroy* operation from the associated channel object to invoke the disconnect operation on all proxies that were created via this *SupplierAdmin* object:

$$\begin{array}{ll}
\textit{Destroy} \mathrel{\widehat=} & (\bigwedge pc : \textit{pushConsumers} \bullet pc.\textit{DisconnectPushConsumer}) \parallel \\
& (\bigwedge pc : \textit{pullConsumers} \bullet pc.\textit{DisconnectPullConsumer})
\end{array}$$

The protected operation *PushToChannel* is for the connected proxies to push data to the associated channel object:

$$\textit{PushToChannel} \mathrel{\widehat=} [\, d? : \mathbb{O} \,] \bullet \textit{channel}.\textit{Transfer}(d?)$$

H.3 *ConsumerAdmin*

The class *ConsumerAdmin* creates the proxy suppliers for consumers and keeps track of these connected proxy suppliers. It defines the first step for connecting consumers to the channel (see the second step defined in classes *ProxyPushSupplier* on page 345 and *ProxyPullSupplier* on page 317); clients use it to obtain proxy suppliers:

$$\begin{array}{|l}
\textit{ConsumerAdmin} \\
\hline
\upharpoonright(\textit{ObtainPushSupplier}, \textit{ObtainPullSupplier}, \\
\quad \textit{DisconnectPushSupplier}, \textit{DisconnectPullSupplier}, \\
\quad \textit{Destroy}, (\textit{PushFromChannel}))
\end{array}$$

$$\begin{array}{l}
channel : \downarrow Channel \\
\Delta \\
pushSuppliers : \mathbb{P} \downarrow ProxyPushSupplier_{\copyright} \\
pullSuppliers : \mathbb{P} \downarrow ProxyPullSupplier_{\copyright}
\end{array}$$

The operation *ObtainPushSupplier* creates and outputs a *ProxyPushSupplier* object. The *ProxyPushSupplier* object can then be used to connect a push-style consumer. The operation *DisconnectPushSupplier* removes the *ProxyPushSupplier* object from *pushSuppliers*:

ObtainPushSupplier

$$\begin{array}{l}
pushSupplier! : \downarrow ProxyPushSupplier \\
\hline
\mathrm{new}(pushSupplier!, ProxyPushSupplier) \\
pushSupplier!.\textsc{Init} \\
pushSupplier!.consumerAdmin = self \\
pushSuppliers' = pushSuppliers \cup \{pushSupplier!\}
\end{array}$$

DisconnectPushSupplier

$$\begin{array}{l}
pushSupplier? : \downarrow ProxyPushSupplier \\
\hline
pushSuppliers' = pushSuppliers \setminus \{pushSupplier?\}
\end{array}$$

The operation *ObtainPullSupplier* creates and outputs a *ProxyPullSupplier* object. The *ProxyPullSupplier* object can then be used to connect a pull-style consumer. The operation *DisconnectPullSupplier* removes the *ProxyPullSupplier* object from *pullSuppliers*:

ObtainPullSupplier

$$\begin{array}{l}
pullSupplier! : \downarrow ProxyPullSupplier \\
\hline
\mathrm{new}(pullSupplier!, ProxyPullSupplier) \\
pullSupplier!.\textsc{Init} \\
pullSupplier!.consumerAdmin = self \\
pullSuppliers' = pullSuppliers \cup \{pullSupplier!\}
\end{array}$$

DisconnectPullSupplier

$$\begin{array}{l}
pullSupplier? : \downarrow ProxyPullSupplier \\
\hline
pullSuppliers' = pullSuppliers \setminus \{pullSupplier?\}
\end{array}$$

The protected operation *Destroy* is called by the *Destroy* operation from the associated channel object to invoke the disconnect operation on all proxies that were created via this *SupplierAdmin* object:

$$\begin{aligned} &Destroy \mathrel{\widehat=} (\bigwedge ps : pushSuppliers \bullet ps.DisconnectPushSupplier) \parallel \\ &\qquad\qquad (\bigwedge ps : pullSuppliers \bullet ps.DisconnectPullSupplier) \end{aligned}$$

The protected operation *PushToChannel* is for the associated channel object to push data to call connected proxy suppliers. Depending on the data delivery model, the proxy suppliers define different interface operations to receive the data. For a push-style proxy supplier, it will directly push the data to the connected consumer; for a pull-style proxy supplier, it will first put the data in a queue and then wait for the client to pull the data from the queue:

$$\begin{aligned} &PushFromChannel \mathrel{\widehat=} \\ &\quad [\, d? : \mathbb{O} \,] \bullet (\bigwedge ps : pushSuppliers \bullet ps.PushToConsumer(d?)) \parallel \\ &\qquad\qquad (\bigwedge ps : pullSuppliers \bullet ps.PushToSupplier(d?)) \end{aligned}$$

Note that the operator $\bigwedge$ conjoins the component operations, the push operations of the proxy suppliers in this case, so that the input data is replicated and pushed to the proxy suppliers.

H.4 *PushSupplier*

A push supplier supports *PushSupplier* interface:

PushSupplier

$$DisconnectPushSupplier \mathrel{\widehat=} [\ ]$$

A concrete push supplier should implement the operation *DisconnectPushSupplier* to terminate the communication. It should release resources used at the supplier to support the communication. The *PushSupplier* object reference should then be disposed. If a *PushSupplier* object is connected to a *PushConsumer* object, invoking the *DisconnectPushSupplier* operation on the *PushSupplier* object should also cause the implementation to call the *DisconnectPushConsumer* operation on the *PushConsumer* object.

H.5 *ProxyPushSupplier*

The class ProxyPushSupplier is first of all a *PushSupplier*. It implements the abstract operation from its super class. It also defines the second step for connecting push consumers to a channel (see the first step in class *ConsumerAdmin* on page 313). It has a state variable *pushConsumer* to keep track of the connected *PushConsumer* object.

ProxyPushSupplier

$$\begin{aligned} &\upharpoonright (ConnectPushConsumer, DisconnectPushSupplier, \\ &\quad (PushToConsumer)) \\ &PushSupplier \end{aligned}$$

$consumerAdmin : \downarrow ConsumerAdmin$
Δ
$pushConsumer : \downarrow PushConsumer_{\circledcirc}$

INIT
$pushConsumer = \mathsf{null}$

The operation *ConnectPushConsumer* connects a *PushConsumer* object to the *ProxyPushSupplier*, but only when it has not been connected to any other *PushConsumer* object:

ConnectPushConsumer
$pushConsumer? : \downarrow PushConsumer$

$pushConsumer = \mathsf{null} \land pushConsumer' = pushConsumer?$

The operation *DisconnectPushSupplier* implements the corresponding abstract operation from its super class. If it is connected to a *PushConsumer* object, the operation disconnects the *PushConsumer* object and invokes the *DisconnectPushConsumer* on the *PushConsumer* object. The *pushConsumer* attribute is then set to null, and the proxy supplier also disconnects itself from *consumerAdmin*:

```
$DisconnectPushSupplier \mathrel{\widehat=}$
    $(\mathbf{if}\ pushConsumer \neq \mathsf{null}$
     $\mathbf{then}\ pushConsumer.DisconnectPushConsumer)\ \|$
    $pushConsumer := \mathsf{null}\ \|$
    $consumerAdmin.DisconnectPushSupplier(self)$
```

The protected operation *PushToConsumer* is for the associated *ConsumerAdmin* object to push data from channel to the connected *PushConsumer* object that has a *Push* interface:

```
$PushToConsumer \mathrel{\widehat=} [d? : \mathbb{O}] \bullet \mathbf{if}\ pushConsumer \neq \mathsf{null}$
                                   $\mathbf{then}\ pushConsumer.Push(d?)$
```

H.6 *PullSupplier*

A pull-style supplier provides the operations defined in the class *PullSupplier* to transmit data:

PullSupplier
$Pull \mathrel{\widehat=} [\,d! : \mathbb{O}\,]$
$TryPull \mathrel{\widehat=} [\,d! : \mathbb{O};\ hasData! : \mathbb{B}\,]$
$DisconnectPullSupplier \mathrel{\widehat=} [\ ]$

A consumer requests data from the supplier by invoking either the operation *Pull* that blocks until the data is available, or the operation *TryPull* which does not block. If the data is available, the operation *TryPull* should output the data $d!$ and set $hasData!$ to true; if the date is not available, *TryPull* should set $hasData!$ to false and the output $d!$ may carry an undefined value.

A concrete pull-supplier should also implement the operation *DisconnectPullSupplier* to terminate the communication. It should release resources used at the supplier to support the communication. The *PullSupplier* object reference should then be disposed. If a *PullSupplier* object is connected to a *PullConsumer* object, invoking the *DisconnectPullSupplier* operation on the *PullSupplier* object should also cause the implementation to call the *DisconnectPullConsumer* operation on the *PullConsumer* object.

H.7 *ProxyPullSupplier*

The class *ProxyPullSupplier* implements the abstract interfaces defined in *PullSupplier*. In addition, it defines the second step for connecting pull consumers to a channel (see the first step in class *ConsumerAdmin* on page 313). It has a state variable *pullConsumer* to keep track of the connected *PullConsumer* object, and a *pending* queue (specified as a sequence) that temporally caches the data pushed from the associated *ConsumerAdmin* object.

ProxyPullSupplier

$\upharpoonright(ConnectPullConsumer, DisconnectPullSupplier,$
$\quad Pull, TryPull, (PushToSupplier))$

$PullSupplier$

$consumerAdmin : \downarrow ConsumerAdmin$
Δ
$pending : \mathrm{seq}\, \mathbb{O}$
$pullConsumer : \downarrow PullConsumer_{\circledcirc}$

INIT

$pullConsumer = \mathsf{null}$

The operation *ConnectPullConsumer* connects a *PullConsumer* object to the *ProxyPullSupplier*, but only when it has not been connected to any other *PullConsumer* object:

ConnectPullConsumer

$pullConsumer? : \downarrow PullConsumer$

$pullConsumer = \mathsf{null} \land pullConsumer' = pullConsumer?$

The operation *DisconnectPullSupplier* implements the corresponding abstract operation from its super class. If it is connected to a *PullConsumer* object, the operation disconnects the *PullConsumer* object and invokes the *DisconnectPullConsumer* on the *PullConsumer* object. The *pullConsumer* attribute is then set to null, and the proxy supplier also disconnect itself from *consumerAdmin*::

$$\begin{array}{l}
\mathit{DisconnectPullSupplier} \mathrel{\widehat=} \\
\quad (\textbf{if}\ \mathit{pullConsumer} \neq \mathsf{null} \\
\quad\ \textbf{then}\ \mathit{pullConsumer}.\mathit{DisconnectPullConsumer}) \ \parallel \\
\quad \mathit{pullConsumer} := \mathsf{null} \ \parallel \\
\quad \mathit{consumerAdmin}.\mathit{DisconnectPullSupplier}(\mathit{self})
\end{array}$$

The operation *Pull* implements the abstract operation *Pull* from its super class according to the required semantics: it blocks the process and keeps trying the operation *TryPull* until there is data available (*hasData*? is true):

$$\begin{array}{l}
\mathit{Pull} \mathrel{\widehat=} \mathit{TryPull} \mathbin{\S} \left[\, d? : \mathbb{O};\ \mathit{hasData}? : \mathbb{B} \,\right] \bullet \\
\qquad\qquad\qquad \textbf{if}\ \mathit{hasData}?\ \textbf{then} \left[\, d! : \mathbb{O} \mid d! = d? \,\right] \textbf{else}\ \mathit{Pull}
\end{array}$$

The operation *TryPull* implements the semantics required in its super class: It checks whether the *pending* queue is empty. If so, sets *hasData*! to false and outputs an arbitrary *d*! from the set $\mathbb{O}$; otherwise, sets *hasData*! to true and outputs the data from the queue according to the First In, First Out (FIFO) order:

$$\begin{array}{|l}
\hline
\mathit{TryPull} \\
d! : \mathbb{O} \\
\mathit{hasData}! : \mathbb{B} \\
\hline
\textbf{if}\ \#(\mathit{pending}) = 0\ \textbf{then}\ \neg\ \mathit{hasData}! \\
\textbf{else}\ \mathit{hasData}! \wedge \mathit{pending} = \langle d! \rangle \frown \mathit{pending}' \\
\hline
\end{array}$$

The protected operation *PushToSupplier* is for the associated *ConsumerAdmin* object to push data from channel to into the *pending* data queue:

$$\mathit{PushToSupplier} \mathrel{\widehat=} \left[\, d? : \mathbb{O} \,\right] \bullet \mathit{pending} := \mathit{pending} \frown \langle d? \rangle$$

H.8 *PushConsumer*

A push-style consumer provides the operations defined in the class *PushConsumer*:

$$\begin{array}{|l}
\hline
\mathit{PushConsumer} \\
\mathit{Push} \mathrel{\widehat=} \left[\, d? : \mathbb{O} \,\right] \\
\mathit{DisconnectPushConsumer} \mathrel{\widehat=} [\ ] \\
\hline
\end{array}$$

A supplier communicates data to the consumer by invoking the operation *Push* which takes the data *d*? as input. The operation *DisconnectPushConsumer* terminates the communication. It should release resources used at the consumer to support the communication. The *PushConsumer* object reference should then be disposed. If a *PushConsumer* object is connected to a *PushSupplier* object, invoking the *DisconnectPushConsumer* operation on the *PushConsumer* object should also cause the implementation to call the *DisconnectPushSupplier* operation on the *PushSupplier* object.

H.9 *ProxyPushConsumer*

The class *ProxyPushConsumer* implements the abstract interfaces defined in *PushConsumer*. It also defines the second step for connecting push suppliers to a channel (see the first step in class *SupplierAdmin* on page 312). It uses a state variable *pushSupplier* to store the reference of the connected *PushSupplier* object.

```
ProxyPushConsumer
  $\upharpoonright(ConnectPushSupplier, DisconnectPushConsumer, Push)$
  $PushConsumer$

    $supplierAdmin : \downarrow SupplierAdmin$
    $\Delta$
    $pushSupplier : \downarrow PushSupplier_{\circledcirc}$

    INIT
    $pushSupplier = \mathsf{null}$
```

The operation *ConnectPushSupplier* connects a *PushSupplier* object to the *ProxyPushConsumer*, but only when it has not been connected to any other *PushSupplier* object:

```
ConnectPushSupplier
  $pushSupplier? : \downarrow PushSupplier$
  ----
  $pushSupplier = \mathsf{null} \wedge pushSupplier' = pushSupplier?$
```

The operation *DisconnectPushConsumer* follows the semantics of the corresponding abstract operation from its super class. If it is connected to a *PushSupplier* object, the operation disconnects the *PushSupplier* object and invokes the *DisconnectPushSupplier* on the *PushSupplier* object. The operation also disconnects this proxy consumer from the *SupplierAdmin* component:

```
$DisconnectPushConsumer \mathrel{\widehat=}$
    $(\textbf{if}\ pushSupplier \neq \mathsf{null}$
     $\textbf{then}\ pushSupplier.DisconnectPushSupplier) \parallel$
    $pushSupplier := \mathsf{null} \parallel$
    $supplierAdmin.DisconnectPushConsumer(self)$
```

The operation *Push* implements the abstract operation from its super class to receive the data from the *PushSupplier* objects and push the data directly to *supplierAdmin* which will in turn push the data into the channel:

$$Push \mathrel{\widehat{=}} [\, d? : \mathbb{O} \,] \bullet supplierAdmin.PushToChannel(d?)$$

H.10 *PullConsumer*

A pull-style consumer should implement the *PullConsumer* interface defined as follows:

$PullConsumer$

$DisconnectPullConsumer \mathrel{\widehat{=}} [\ ]$

Any concrete pull-consumer inheriting *PullConsumer* should implement the operation *DisconnectPullConsumer* to terminate the communication and release resources occupied at the consumer to support the communication. The *PullConsumer* object reference should then be disposed. If a *PullConsumer* object is connected to a *PullSupplier* object, invoking the *DisconnectPullConsumer* operation on the *PullConsumer* object should also cause the implementation to call the *DisconnectPullSupplier* operation on the *PullSupplier* object.

H.11 *ProxyPullConsumer*

The class ProxyPullConsumer implements the abstract operation from its super class *PullConsumer*. It also defines the second step for connecting push suppliers to a channel (see the first step in class *SupplierAdmin* on page 312). It has a state variable *pullSupplier* to store the reference of the connected *PullSupplier* object.

$ProxyPullConsumer$

$\upharpoonright(ConnectPullSupplier, DisconnectPullConsumer)$

$PullConsumer$

$supplierAdmin : \downarrow SupplierAdmin$

Δ

$pullSupplier : \downarrow PullSupplier_{\circledcirc}$

$\textsc{Init}$

$pullSupplier = \mathsf{null}$

It provides an interfacing operation *ConnectPullSupplier* for a *PullSupplier* object to be connected to this proxy consumer:

$$\begin{array}{|l}
\textit{ConnectPullSupplier} \\
\hline
\textit{pullSupplier?} : \downarrow \textit{PullSupplier} \\
\hline
\textit{pullSupplier} = \mathsf{null} \land \textit{pullSupplier}' = \textit{pullSupplier?}
\end{array}$$

It implements the abstract operation *DisconnectPullConsumer* from its super class according to the required semantics:

$$\begin{array}{l}
\textit{DisconnectPullConsumer} \mathrel{\widehat=} \\
\quad (\textbf{if}\ \textit{pullSupplier} \neq \mathsf{null} \\
\quad\ \ \textbf{then}\ \textit{pullSupplier}.\textit{DisconnectPullSupplier}) \ \| \\
\quad \textit{pullSupplier} := \mathsf{null} \ \| \\
\quad \textit{supplierAdmin}.\textit{DisconnectPullConsumer}(\textit{self})
\end{array}$$

The private operation *PullFromSupplier* tries to pull data from the connected *pullSupplier*. If there is data available, it pulls the data and then pushes to the associated *supplierAdmin* which will in turn push the data into the channel:

$$\begin{array}{ll}
\textit{PullFromSupplier} \mathrel{\widehat=} & [\textit{pullSupplier} \neq \mathsf{null}] \bullet \\
& \textit{pullSupplier}.\textit{tryPull} \fatsemi \\
& [\textit{hasData?} : \mathbb{B} \mid \textit{hasData?}] \land \\
& \textit{supplierAdmin}.\textit{PushToChannel}
\end{array}$$

A *ProxyPullConsumer* object is an active object. It has an active process that keeps trying to pull data from the connected supplier on behalf of the channel. Providing with an active *ProxyPullConsumer* to the suppliers, the channel appears to be active and plays the role of an *agent* (if the channel pushes data to the passive consumer) or a *procure* (if the consumer pulls the data from the channel).

$$\begin{array}{l}
\textit{IdleTick} \mathrel{\widehat=} [\neg\ \text{pre}\ \textit{PullFromSupplier} \land \tau' = \tau + 1] \\
\textsc{Process} \mathrel{\widehat=} \mu\, P \bullet (\textit{PullFromSupplier} \ [\!]\ \textit{IdleTick}) \fatsemi P
\end{array}$$

H.12 An example of the push model

As an example, this section shows how a *PushSupplier ps* delivers data $d : \mathbb{O}$ to a *PushConsumer pc* through a global channel c : *Channel* using a canonical push model as shown in figure 7.4(a) on page 84. At the consumer *pc* side, *pc* implements the *Push* interface. Let's first get the *ConsumerAdmin* object from the *Channel c* by invoking the operation *c.ForConsumer*. Let *consumerAdmin* denote the output from this operation.

The consumer *pc* then obtains a *ProxyPushSupplier* object from *consumerAdmin*: *consumerAdmin.ObtainPushSupplier*. A *ProxyPushSupplier* is created and added to the push supplier list by *consumerAdmin*. Let *proxyPushSupplier* denote the created *ProxyPushSupplier* object. The consumer *pc* can be connected to the proxy: *proxyPushSupplier.ConnectPushConsumer(pc)*. The consumer *pc* is then ready awaiting data being pushed through its *Push* operation by *proxyPushSupplier*.

At the supplier side, the *PushSupplier* ps can be connected to the *Channel* c in a similar manner: first invoke $c.ForSupplier$ to get the *SupplierAdmin* object (denoted as *supplierAdmin*), then obtain a *ProxyPushConsumer* object (denoted as *proxyPushConsumer*) by invoking $supplierAdmin.ObtainPushConsumer$ and connect ps to the proxy consumer by calling $proxyPushConsumer.ConnectPushSupplier(ps)$.

To push the data d though, ps may simply invoke the operation $proxyPushConsumer.Push(d)$. The object *proxyPushConsumer* then pushes the data into the channel by the operation $supplierAdmin.PushToChannel$ and the channel *Transfer*s the data to *consumerAdmin*

$$[\, d? : \mathbb{O} \,] \bullet consumerAdmin.PushFromChannel(d?).$$

The operation $consumerAdmin.PushFromChannel$ then pushes the data to all registered *ProxyPushSupplier* objects. Notice that the consumer pc is connected to *proxyPushSupplier* and *proxyPushSupplier* is registered to *consumerAdmin*. The operation *PushToConsumer* of *proxyPushSupplier* is then invoked and in turn, the operation $pc.Push$ is finally invoked to receive the data.

BACKGROUND MATERIAL I

Specifications of the Real-time Channel Pattern

This background material presents the formal specification of the components of the Real-time Channel pattern that is introduced in section 7.3 of chapter 7.

I.1 *QoSProperties*

The type *QoSProperties* is a list of $name \mapsto value$ pairs that defines the all possible QoS properties, such as reliability and priority. Let's model these $name \mapsto value$ pairs as a *Dictionary* as it is specified on on page 265:

$$QoSProperties == Dictionary$$

Table I.1 on the following page lists a number of property names and their possible values that are supported in the IPML system design. The meanings of these names and values will be explained later as they are encountered, otherwise one may also refer to the CORBA notification service specification(OMG, 2004b) as similar QoS properties are also recommended in this specification.

The *QoSProperties* are designed as a set of $name \mapsto value$ pairs instead of a structurally equivalent data type that includes the properties as strictly typed attributes. While the later is straightforward, it is clear that whatever choices of properties and their permitted values are, it is not possible to cover all use cases. The former enables implementations to extend the properties and to facilitate simple evolution of QoS properties. Implementations may add the properties understood by a particular QoS management, for example in our case of Real-time Channel pattern, by the *Scheduler* (see definition on page 327).

The QoS requirements can be set at different levels, from per data transmission to every connected proxy, or to the channel. Note that setting certain QoS properties to a particular level can be meaningless. For example, it makes no sense to set

Table I.1: QoS properties

Property	Type	data	Proxy	Channel
reliability	*Reliability* ::= BestEffort | Persistent	✓		✓
priority	$\mathbb{N}$	✓	✓	✓
deadline	$\mathbb{T}$	✓		
timeout	$\mathbb{T}$	✓	✓	✓
queueSize	$\mathbb{N}$			✓
orderPolicy	*OrderPolicy* ::= FIFO |Deadline			✓
discardPolicy	*DiscardPolicy* ::= FIFO | LIFO			✓

deadline (absolute expiry time, e.g. 12:00pm, woensdag 30 augustus 2006 om 16.00 uur) for data transmission on the entire channel. Table I.1 also summarizes which QoS properties can be set at each level. Also note that the channel and proxy QoS properties set the default values for the data. If the data carries a certain QoS property, it overrides the default value defined in the proxy or the channel.

I.2 *Filter*

The Real-time Channel pattern first adds a Filter object to the Channel class in the Channel pattern. It allows consumers to subscribe for particular subset of data. The channel then uses these subscriptions to filter the data from suppliers, only forwarding them to interested consumers.

Integrating the filters in the channels relieves consumers from implementing filtering semantics. It also reduces communication load by eliminating filtered data to appear in the channels instead of eliminating them at consumers. Filtering might also be implemented at the suppliers, however, this requires the suppliers to have knowledge of the consumers, hence sacrifice the benefits of decoupling consumers and suppliers, which is one of the primary motivations of the Channel pattern.

Filtering requires a well defined type naming system for the objects. It is required that any data object sent through the channels must implement the *TypedData* interface attributes to identify its source (the supplier), the type name and the QoS requirements:

TypedData

sourceid : *String*
typename : *String*
qos : *QoSProperties*

The type names identify different types of the data objects and the type naming system is shared by all the parties involved in the communication. The element *sourceid* indicates the global identity of the source supplier object. The get the global identity of an object, let's define a global function $\__{\textcircled{\scriptsize id}}$:

$$\__{\textcircled{\scriptsize id}} : \mathbb{O} \rightarrowtail String$$

The consumers interests about the coming $\downarrow TypedData$ is modeled as *DataInterest*

$$\begin{array}{|l}
\hline
\textit{DataInterest} \\
\begin{array}{|l|}
\hline
sources : String \\
types : String \\
qos : QoSProperties \\
\hline
\end{array} \\
\hline
\end{array}$$

where the attributes *sources* and *types* describe the interested data sources and types using for example regular expressions, and *qos* describes the minimum *qos* requirements. For sources and types an operator "matches" can be defined to see whether a source or a type matches the interests described using the regular expressions[1]

$$\_ \text{ matches } \_ : String \times String \rightarrow \mathbb{B}$$

while for *QoSProperties* the operator "⇒" is used to indicate one QoS requirement is "stronger" than another:

$$\_ \Rightarrow \_ : QoSProperties \times QoSProperties$$

where $qos_1 \Rightarrow qos_2$ means that qos_2 is always satisfied wherever qos_1 is satisfied.

An operator is then defined to see wether an object of $\downarrow TypedData$ is "in" the interests described in an object of *DataInterest*:

$$\begin{array}{|l}
\_ \text{ in } \_ : \downarrow TypedData \times DataInterest \\
\hline
\forall\, d : \downarrow TypedData;\ i : DataInterest \bullet \\
\quad d.sourceid \text{ matches } i.sources \land \\
\quad d.typename \text{ matches } i.types \land \\
\quad d.qos \Rightarrow i.qos
\end{array}$$

A *Filter* object maintains a registry of pairs: *RTProxyPushSupplier* objects (on behalf of the connected consumers), and their subscribed interests. The read-only attribute *subscribedProxies* is the set of the proxies that are currently in the registry. The registry is initialized as an empty set:

$$\begin{array}{|l}
\hline
\textit{Filter} \\
\upharpoonright((subscribedProxies, Subscribe, Unsubscribe, Filtrate))
\end{array}$$

[1]An alternative, slightly more abstract formalization would be to define types as a set, e.q. $types : \mathbb{P}\,\mathbb{O}$. The present description is closer to the practical implementation.

$$\begin{array}{|l}
\Delta \\
registry : \downarrow RTProxyPushSupplier \leftrightarrow \mathbb{P}\, DataInterest \\
subscribedProxies : \mathbb{P} \downarrow RTProxyPushSupplier \\
\hline
subscribedProxies = \mathrm{dom}(registry)
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textsc{Init}} \\
\hline
registry = \varnothing
\end{array}$$

The *Filter* object also provides operations to *Subscribe* and *Unsubscribe* the pairs, and an operation *RemoveProxy* to remove all pairs that is related to a particular proxy:

$$\begin{array}{l}
Subscribe \mathrel{\widehat=} \\
\quad [\,proxy? : \downarrow RTProxyPushSupplier; \\
\qquad interests?, interests : \mathbb{P}\, DataInterest\,] \bullet \\
\quad \mathbf{if}\, proxy? \mapsto interests \in registry \\
\quad \mathbf{then}\, registry := (registry \oplus \{proxy? \mapsto (interests \cup interests?)\}) \\
\quad \mathbf{else}\, registry := (registry \oplus \{proxy? \mapsto interests?\}) \\
Unsubscribe \mathrel{\widehat=} \\
\quad [\,proxy? : \downarrow RTProxyPushSupplier; \\
\qquad interests?, interests : \mathbb{P} \downarrow TypedData\,] \bullet \\
\quad \mathbf{if}\, i? \mapsto d \in registry \\
\quad \mathbf{then}\, registry := (registry \oplus \{proxy? \mapsto (interests \setminus interests?)\}) \\
RemoveProxy \mathrel{\widehat=} [\,proxy? : \downarrow RTProxyPushSupplier\,] \bullet \\
\qquad\qquad registry := \{proxy?\} \lhd registry
\end{array}$$

Given a particular instance of $\downarrow TypedData$, the following operation finds all *RTProxyPushSupplier* objects (hence connected consumers) in the registry that are interested in this data:

$$\begin{array}{|l}
\multicolumn{1}{l}{Filtrate} \\
d? : \downarrow TypedData \\
interestedProxies! : \mathbb{P} \downarrow RTProxyPushSupplier \\
\hline
interestedProxies! = \\
\quad \mathrm{dom}(registry \rhd \\
\quad \{interests : \mathrm{ran}(registry) \mid (\exists\, i : interests \bullet d? \text{ in } i)\})
\end{array}$$

When data arrives from the suppliers, the *Filter* object determine which supplier proxies (on behalf of the consumers) should receive the data. It forwards the data to the *Dispatcher* object, which handles the details of dispatching each data to its supplier proxies in accordance with the priority of the proxy/data tuple.

I.3 *Scheduler*

Given an input of $d? : \downarrow TypedData$ and a target $proxy? : \downarrow RTProxyPushSupplier$, the Dispatcher collaborates with the runtime *Scheduler* to determine priority values:

```
Scheduler
  $\upharpoonright((channel, Schedule))$

  $channel : \downarrow RTChannel$

  Schedule
    $proxy? : \downarrow RTProxyPushSupplier$
    $d? : \downarrow TypedData$
    $priority!, p : \mathbb{N}_{\circledcirc}$
    $order! : \mathbb{N}_{\circledcirc}$
    $t : \mathbb{T}$                    [ $p$ and $t$ are existential temporary variables]
    ----
    Determine the QoS priority:
    if $Get(d?.qos, \text{“}priority\text{”}) = p$
    then $priority! = p$
    else if $Get(proxy?.qos, \text{“}priority\text{”}) = p$
        then $priority! = p$
        else if $Get(channel.qos, \text{“}priority\text{”}) = p$
            then $priority! = p$
            else $priority! = \text{null}$
    Determine the dispatching order:
    if  $Get(d?.qos, \text{“}orderPolicy\text{”}) = \text{Deadline}$
    then  if $Get(d?.qos, \text{“}deadline\text{”}) = t$
          then $order! = t$
          else if $Get(d?.qos, \text{“}timeout\text{”}) = t$
              then $order! = \tau + t$
              else if $Get(proxy?.qos, \text{“}timeout\text{”}) = t$
                  then $order! = \tau + t$
                  else if $Get(channel.qos, \text{“}timeout\text{”}) = t$
                      then $order! = \tau + t$
                      else $order! = \text{null}$
    else if $Get(channel.qos, \text{“}orderPolicy\text{”}) = \text{FIFO}$
        then $order! = \tau$
        else $order! = \text{null}$
```

RTChannel extends the class *Channel* in the Channel Pattern with QoS properties and other real-time extensions (see definition on page 334).

The reason for decoupling the *Scheduler* from the data dispatching task is to allow scheduling policies to evolve independently of the dispatching mechanism. The operation *Schedule* is an example implementation from the IPML system implementation. It uses dynamic scheduling policies such as Priority, Deadline and FIFO(First-in First-out). The algorithm to determine the primary preemption priority and the order in the same preemption priority depends on the implementation. By separating the responsibilities of scheduling from dispatching, the *Scheduler* can be replaced without affecting unrelated components in the channel.

The design of the class *Scheduler* here implements a *supplier-consumer override* strategy to determine the primary preemption priority. If the supplier defines the priority of a particular transaction in the pushed data, otherwise if the consumer has set the priority through its connected supplier proxy, the priority settings of the Channel is overridden. This particular strategy is specified here as an example implementation because in the IPML system, top-down timing control commands have the highest priority and this strategy is often applied. But not saying that this is the only strategy to be applied in the system. In some cases, for instance if the data consumer presents a user interface and a certain interested subset of data must be immediately presented to the user, a *consumer-channel override* strategy may apply.

The dispatching order in the same preemption priority is determined according to the channel's *OrderPolicy*. Instead of trying to specify every possible order policy, the operation *Schedule* here only shows an example algorithm. If the channel's *OrderPolicy* is Deadline, the algorithm checks the *timeout* QoS requirements of the data itself, the consumer (the supplier proxy), otherwise the channel. Again as an example, a *supplier-consumer override* strategy is applied here. The value of the *order*! output records the required finishing time. The data that requires the earliest finishing time will be dispatched first. If the channel's *OrderPolicy* is FIFO, the value of the *order*! output records the arriving time of the data, and the data that arrives earliest will be dispatched first.

When the preemption priority or the order is specified by no one, the result is set to null – which means the priority or the order is undetermined by the *Scheduler*.

I.4 *PriorityQueue*

The *RTChannel* maintains a sequence of *PriorityQueue* objects, one for each possible preemption priority (see definition on page 334). The *PriorityQueue* has an attribute *priority* to associate itself to a preemption priority set by the *RTChannel*, and an attribute *maxsize* to limit its maximum number of elements. If there is a QoS requirement on the maximum queue size, the *maxsize* should be set accordingly. The elements of the queue are the tuples of the targeted proxy, the data to be delivered, and the dispatching order. In the following specification, the attributes *isEmpty* and *isFull* reflect whether the queue is empty, or the number of the elements has reached the *maxsize*:

PriorityQueue

$\upharpoonright((channel, priority, isEmpty, isFull, Enqueue, Dequeue))$

$$
\begin{array}{|l}
channel : \downarrow RTChannel \\
priority, maxsize : \mathbb{N} \\
isEmpty, isFull : \mathbb{B} \\
\Delta \\
queue : \mathrm{seq}(\downarrow RTProxyPushSupplier \times \downarrow TypedData \times \mathbb{N}) \\
\hline
queue = \langle\, \rangle \Leftrightarrow isEmpty \\
\mathbf{let}\; s == Get(channel.qos, \text{“}queueSize\text{”}) \bullet s \neq \mathsf{null} \Rightarrow size = s \\
\# queue = maxsize \Leftrightarrow isFull
\end{array}
$$

The *PriorityQueue* acts as a normal queue when dequeuing the elements – the tuple at the head of queue always get dequeued first. The result of the *Dequeue* operation outputs the targeted proxy and the data. The dispatching order is only needed when enqueuing the tuples so that the elements are kept in the queue in the required dispatching order. When dequeuing, the order is no longer needed by other components hence it is dropped from the output:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\; Dequeue\; \_\_\_\_} \\
proxy! : \downarrow RTProxyPushSupplier \\
d! : \downarrow TypedData \\
n : \mathbb{N} \\
\hline
queue = \langle (proxy!, d!, n) \rangle \frown queue'
\end{array}
$$

When enqueuing an element, the operation *Enqueue* first checks whether there is still space left in the queue. If the queue *isFull*, it invokes the operation *Discard* to remove an element from the queue and give the space to the new element, so that the actual enqueuing operation $Enqueue_0$ can be invoked without exceeding the *maxsize* limit:

$$Enqueue \mathrel{\widehat{=}} (\mathbf{if}\; isFull\; \mathbf{then}\; Discard) \mathbin{⨾} Enqueue_0$$

The operation *Discard* drops an element from the queue according to the QoS requirement of the channel. If the *DiscardPolicy* of the channel QoS requirement is set to LIFO (Last-In First-Out), the element at the tail of the queue is dropped, otherwise a FIFO (First-In First-Out) policy is followed – the element at the head of the queue is dropped instead. The *Discard* drops elements silently – there is no output from this operation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\; Discard\; \_\_\_\_} \\
proxy : \downarrow RTProxyPushSupplier \\
d : \downarrow TypedData \\
n : \mathbb{N} \\
\hline
\end{array}
$$

```
if $Get(channel.qos, “DiscardPolicy”) = \mathsf{LIFO}$
then $queue = queue' \frown \langle (proxy, d, n) \rangle$
else $queue = \langle (proxy, d, n) \rangle \frown queue'$
```

The *PriorityQueue* has a special queuing behavior comparing to a normal FIFO queue. The elements are not always enqueued to the end of queue. Instead, the elements are inserted into the queue according to the dispatching order parameter so that the *Scheduler* can use this parameter to control the actual dispatching order. A lower number to the order parameter gives closer position to the head of queue.

```
Enqueue_0
$proxy? : \downarrow RTProxyPushSupplier$
$d?, d : \downarrow TypedData$
$order? : \mathbb{N}$
$t : \mathbb{T}$
----
$d.source = d?.source \wedge d.type = d?.type \wedge d.data = d?.data$
Set the absolute deadline if possible:
if $Get(d?.qos, “deadline”) = t$
then $d.qos = d?.qos$
else if $Get(d?.qos, “timeout”) = t$
    then $d.qos = d?.qos$
    else if $Get(proxy?.qos, “timeout”) = t$
        then $d.qos = Put(d?.qos, “deadline”, \tau + t)$
        else if $Get(channel.qos, “timeout”) = t$
            then $d.qos = Put(d?.qos, “deadline”, \tau + t)$
            else $d.qos = d?.qos$
Insert into the queue according to $order?$:
if $order? = \mathsf{null}$
then if $isEmpty$ then $queue' = \langle (proxy?, d, 0) \rangle$
     else $\exists (p, d, n) : \{\mathrm{last}(queue)\} \bullet$
          $queue' = queue \frown \langle (proxy?, d, n) \rangle$
else $queue' =$
    $queue \upharpoonright \{(p, d, n) : \mathrm{ran}(queue) \mid n \leqslant order?\} \frown$
    $\langle (proxy?, d, order?) \rangle \frown$
    $queue \upharpoonright \{(p, d, n) : \mathrm{ran}(queue) \mid n > order?\}$
```

When a tuple element is inserted into the queue, the operation $Enqueue_0$ also tries to update the relative “*timeout*” requirement to an absolute deadline if the data does not require a deadline explicitly. Again the *supplier-consumer override* strategy is applied. The relative “*timeout*” requirements of the data, the proxy and the channel are checked in turn and the first “*timeout*” found is converted to an absolute deadline property for the data, according to the current system time. When dispatching, the *Dispatcher* will decide what to do if a data has missed its deadline. The filtering operator $\upharpoonright$ filters a sequence s with a set v so that $s \upharpoonright v$ contains just those elements of s which are members of v, in the same order as in s.

I.5 *Dispatcher*

The *Dispatcher* is responsible for dispatching the queued data to targeted supplier proxies. The proxies will then forward the data to the connected consumers. Depending on the placement of the data in the priority queues (A queue of ordered priority queues maintained by the *Channel*, see definition on page 334), the *Dispatcher* may preempt a running thread to dispatch the data to the paired proxy, according to the priority of the queue. Inside the same queue, the *Dispatcher* always dispatches the data from the head so that the scheduled dispatching order is ensured.

Preemption according to the priority is an important mechanism of the *Dispatcher*. Most of the real-time scheduling policies require preemption. For example if a process P_2 of priority of 2 is running when another process P_1 with priority of 1 becomes runnable, P_1 should be preempted so that P_2 is suspended and P_1 can run immediately, when P_1 and P_2 share certain hardware or software resources, for example a single Central Processing Unit (CPU) in a system. P_2 can continue when processes of priority of 1 completes, unless it is preempted to a priority of 0.

One of the design decisions here is to separate the functionality of data dispatching from the priority queues. This allows the implementation of the Dispatcher to change independently of other channel components, so that different dispatching mechanisms can be implemented. The Dispatcher may implement for example the following three preemption strategies:

Single thread dispatching Without the support for multi-thread dispatching, a signal thread can be used used to dispatch data based on priority. In this model, all priority queues are actually concatenated into one queue with higher priority queues in the front. However, once a dispatching process starts, the consumer can run to completion regardless of the arrival of higher priority data. The dispatching process has to wait until the consumer finishes. As a result, the channel becomes a synchronous data delivery mechanism for suppliers.

Single thread dispatching with deferred preemption This also a single-thread implementation where on thread is responsible for dispatching all queued data. However it requires the consumers cooperatively preempt their data receiving processes according to their own priority preferences, and the priority preference of the data supplier is ignored. The benefit of this “deferred preemption” is its ability of reducing the context switching, synchronization and data movement incurred by preemptive multi-threading implementations. However, preemption is deferred to the extent that the consumers need to check to see whether they need to preempt their receiving processing. In a distributed setting, “deferred preemption” can only be effective among consumers that reside in the same local system, but not across the systems.

Preemptive multi-thread dispatching Multi-threading is supported by an increasing number of the operating systems. The *Dispatcher* dispatches the data from the head of a priority queue by preempt a thread with a preemption priority that is corresponding to the priority of the queue, and then leaves the preemption task to the underlying multi-thread operating system. The advantage of this model is that the *Dispatcher* can take the advantage of the operating system support for preemption by associating appropriate operating system priorities to each thread. When a thread at the higher priority becomes ready to run, the operating system will preempt any lower priority running thread and allow the higher priority thread to run. The disadvantage is that multi-threading incurs context switching overhead. Furthermore, the implementation must synchronize the access to the resources that can be shared by multiple threads.

The following specification is an implementation of preemptive multi-thread dispatching. The *Dispatcher* is an active component and it has a process that keeps polling the priority queues in a high-priority first order:

```
Dispatcher
$\upharpoonright((channel))$

    $channel : \mathbb{P}\downarrow RTChannel$

$Poll \mathrel{\widehat=} [\, pqq? : \text{seq}\ PriorityQueue \,] \bullet$
        $\textbf{if}\ pqq? \neq \langle\rangle$
        $\textbf{then}\ [\, pq : PriorityQueue \mid pq = \text{head}(pqq) \,] \bullet$
            $\textbf{if}\ pq.isEmpty\ \textbf{then}\ Pool(\text{tail}(pqq?))$
            $\textbf{else}\ pq.Dequeue \wedge$
                $[\, dt : DispatchingThread \mid$
                    $\text{new}(dt, DispatchingThread) \wedge$
                    $dt.channel = channel \wedge$
                    $dt.preemption = pq.priority \,] \, ⨟$
                $dt.Dispatch$
$\textsc{Process} \mathrel{\widehat=} \mu\, D \bullet Poll(channel.queues) \, ⨟ \, D$
```

The operation *Poll* recursively polls the channel priority queues until the first non-empty queue is located. The data at the head of this queue is dispatched by creating and preempting a *DispatchingThread*. The preemption priority of the thread set according to the priority of the queue. The operation *Poll* does not traverse all the priority queues in one loop. Instead, it will start over again from the beginning of the queue after a top priority and top ordered data is found and dispatched to interested proxies. This ensures if a data with highest priority and order is inserted into the queues during this process, the data will be dispatched first in the next round immediately.

Object-Z does not have a formal mechanism to specify the preemption priority of an active process or thread. The following specification of the *DispatchingThread* defines an attribute *preemption*. Let's assume that the underlying platform would take this attribute as the preemptive parameter and preempt the process accordingly. The

DispatchingThread is an active process. After initialization, it waits for other process to call its *Dispatch* operation to set the data to be dispatched *d* and the targeted *proxy*:

DispatchingThread

$\upharpoonright((preemption, channel, Dispatch))$

$$\begin{array}{l} channel : \downarrow RTChannel \\ preemption : \mathbb{N} \\ \Delta \\ proxy : \downarrow RTProxyPushSupplier_{\circledcirc} \\ d : \downarrow TypedData_{\circledcirc} \end{array}$$

INIT

$$proxy = \mathsf{null} \land d = \mathsf{null}$$

Dispatch

$$\begin{array}{l} proxy? : \downarrow RTProxyPushSupplier \\ d? : \downarrow TypedData \\ t : \mathbb{T} \\ \hline \textbf{if } Get(d?.qos, \text{``}deadline\text{''}) = t \land \tau > t \land \\ \quad Get(channel.qos, \text{``}reliability\text{''}) \neq \mathsf{BestEffort} \\ \textbf{then } \mathrm{true} \qquad [\,\mathsf{Deadline\ missed,\ does\ nothing}] \\ \textbf{else } proxy' = proxy? \land d' = d? \land started' \end{array}$$

$$\begin{array}{l} IdleTick \mathrel{\widehat{=}} [\,\tau' = \tau + 1\,] \\ \textsc{Process} \mathrel{\widehat{=}} \mu\, DT \bullet \textbf{if } proxy = \mathsf{null} \lor d = \mathsf{null} \\ \qquad\qquad \textbf{then} (IdleTick \mathbin{⨟} DT) \textbf{ else } proxy.Push(d) \end{array}$$

The operation *Dispatch* first checks the QoS reliability requirement. If it is required to deliver the data on the best effort basis and unfortunately the deadline is missed, the operation *Dispatch* simply does nothing, otherwise it sets the attributes *proxy* and *d* for delivery.

The active PROCESS operation recursively calls *IdleTick* until both attributes *proxy* and *d* are set, than invokes the *proxy*'s *Push* operation to dispatch the data *d*. After it is done, the active process terminates. Whether and how a terminated thread is treated depend on the underlying platform. In C and C++ based implementations, it is the programmer's responsibility to recycle the resources occupied by these dead threads. In Java virtual machines, it will be collected by the garbage collector and swept out from the memory. In the implementation, the Thread class from the standard java.lang library is extended to realize *DispatchingThread*, hence let's leave the resource recycling task to the garbage collector.

I.6 *RTChannel*

The *RTChannel* extends the *Channel* component from the Channel pattern with the operation *Transfer* rewritten to meet the QoS requirements, and the *ConsumerAdmin* object is required to be a *RTConsumerAdmin* object to manage real-time push proxies:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{RTChannel}} \\
\upharpoonright(\cdots) \\
\mathit{Channel}[\mathit{Transfer}_c / \mathit{Transfer}]
\end{array}
$$

The following declares two constants: LowestPriority – the lowest priority, and DefaultPriority if the *Scheduler* can not calculate a valid priority that is within the range of $0 \mathinner{..} \mathsf{LowestPriority}$. Without loss of generality, in this specification the valid priorities are modeled as a continuous subset of natural numbers, starting from 0 as the highest priority to the LowestPriority:

$$
\begin{array}{|l}
\mathsf{LowestPriority} : \mathbb{N} \\
\mathsf{DefaultPriority} : 0 \mathinner{..} \mathsf{LowestPrority} \\
\hline
\mathsf{LowestPriority} > 0
\end{array}
$$

Every *RTChannel* has its own *qos* requirements, and these requirements could be overridden in a specific transaction if the suppliers or the consumers have a different requirement and if a “supplier-consumer override” strategy is applied. To manage the transactions to meet the *qos* requirements, *Filter*, *Sheduler* and *Dispatcher* objects are used, together with a list of priority queues, one for each valid priority. Since the priorities are modeled as natural numbers, it is convenient to manage these queues in an indexed list, that is, a sequence of *PriorityQueue* objects:

$$
\begin{array}{|l}
\mathit{qos} : \mathit{QoSProperties} \\
\mathit{filter} : \downarrow \mathit{Filter}_{\copyright} \\
\mathit{scheduler} : \downarrow \mathit{Scheduler}_{\copyright} \\
\mathit{dispatcher} : \downarrow \mathit{Dispatcher}_{\copyright} \\
\mathit{priorityQueues} : \mathrm{seq}\ \mathit{PriorityQueue}_{\copyright} \\
\mathit{consumerAdmin} : \downarrow \mathit{RTConsumerAdmin}_{\copyright} \\
\hline
\mathit{scheduler}.\mathit{channel} = \mathit{dispatcher}.\mathit{channel} = \mathit{self} \\
\#(\mathit{priorityQueues}) = \mathsf{LowestPriority} + 1 \\
\forall\, n \mapsto q : \mathit{priorityQueues} \bullet q.\mathit{priority} = n - 1 \land q.\mathit{channel} = \mathit{self}
\end{array}
$$

The operation *Transfer* accepts objects of any type as input. If it is not an object of the type *TypedData*, the data will be sent though the facilities inherited from the Channel pattern, otherwise through the real-time QoS controlling facilities. The *Filter* finds all subscribed proxies that are interested in this input. For each proxy, the operation then invokes the operation *ScheduleEnqueue* to calculate the priority and the dispatching order, and enqueue the data and the proxy for dispatching. For other push suppliers that are connected to *consumerAdmin* but not subscribed to the filter at all, this operation invokes *ScheduleEnquene* as if these suppliers have subscribed to all possible data input:

```
$Transfer \mathrel{\widehat=} [\, d? : \mathbb{O} \,] \bullet$
    $\textbf{if}\ d? \notin {\downarrow}TypedData\ \textbf{then}\ Transfer_c$
    $\textbf{else}\ (filter.Filtrate(d?) \bullet \bigwedge p : interestedProxies! \bullet$
            $ScheduleEnqueue(proxy \leadsto p, d \leadsto d?)$
        $) \setminus (interestedProxies!) \ \parallel$
        $(\bigwedge p : consumerAdmin.pushSuppliers \setminus filter.subscribedProxies \bullet$
            $ScheduleEnqueue(proxy \leadsto p, d \leadsto d?)$
```

The next two internal operations are specified separately but they are actually parts of the operation *Transfer*, otherwise the operation *Transfer* would be hard to read. The operation *ScheduleEnqueue* first invoke the *Schedule* operation from the *Scheduler* to calculate the preemption priority and the dispatching order within the same priority for a pair of proxy and data, then “regulates” the priority in case the priority could not be defined by the *Scheduler*, or in case the calculated priority is not valid. The data, together with its targeted proxy are then enqueued into the priority queue, awaiting for the *Dispatcher* to do the rest.

```
$ScheduleEnqueue \mathrel{\widehat=}$
    $[\, proxy? : {\downarrow}RTProxyPushSupplier;\ d? : {\downarrow}TypedData \,] \bullet$
        $scheduler.Schedule(proxy \leadsto proxy?, d \leadsto d?) \mathbin{⨾}$
        $RegulatePriority \mathbin{⨾}$
        $([\, priority? : \mathbb{N};\ order? : \mathbb{N}_{\circledcirc};\ queue : priorityQueues \mid$
            $queue.priority = priority? \,] \bullet$
            $queue.Enqueue(proxy \leadsto proxy?, d \leadsto d?, order \leadsto order?)$
```

The operation *RegulatePriority* checks whether the input *proirity*? is valid and returns a valid *priority*!. If the input is null, it is replaced with the DefaultPriority. If it is out of the range of $0..LowestPriority$, the operation returns the closest valid priority:

```
RegulatePriority
    $priority? : \mathbb{N}_{\circledcirc}$
    $priority! : \mathbb{N}$
    ----------------
    $\textbf{if}\ priority? = \mathsf{null}\ \textbf{then}\ priority! = \mathsf{DefaultPriority}$
    $\textbf{else if}\ priority? > \mathsf{LowestPriority}$
        $\textbf{then}\ priority! = \mathsf{LowestPriority}$
        $\textbf{else if}\ priority? < 0\ \textbf{then}\ priority! = 0$
            $\textbf{else}\ priority! = priority?$
```

Note that the *RTChannel* does not directly push data through the *ConsumerAdmin* to the supplier proxies as it does in the Channel pattern (see specification on page 311. Instead it enqueues the data for the active Dispatcher to dispatch it later.

I.7 *RTConsumerAdmin*

The extensions of *RTConsumerAdmin* to *ConsumerAdmin* are straightforward. The *channel* is changed to the type of *RTChannel* so that its *Filter* is accessible for *RTConsumberAdmin*, and further for the connected proxies and consumers to subscribe or unsubscribe their interests:

RTConsumerAdmin

$\upharpoonright(\cdots, (\mathit{pushSuppliers}, \mathit{SubscribeToFilter})$

$\mathit{ConsumerAdmin}[\mathit{ObtainPushSupplier}_{ca} / \mathit{ObtainPushSupplier}]$

$\mathit{channel} : \downarrow \mathit{RTChannel}$

The operation *ObtainPushSupplier* from *ConsumerAdmin* is overridden to create and output RTProxyPushSupplier objects instead of *ProxyPushSupplier* objects:

ObtainPushSupplier

$\mathit{pushSupplier}! : \downarrow \mathit{RTProxyPushSupplier}$

$\mathrm{new}(\mathit{pushSupplier}!, \mathit{RTProxyPushSupplier})$

$\mathit{pushSupplier}!.\textsc{Init}$

$\mathit{pushSupplier}!.\mathit{consumerAdmin} = \mathit{self}$

$\mathit{pushSuppliers}' = \mathit{pushSuppliers} \cup \{\mathit{pushSupplier}!\}$

The next three operations delegates the interest subscription and unsubscription from the proxies to the channel's filter object:

$\mathit{SubscribeToFilter} \mathrel{\widehat=} \mathit{channel.filter.Subscribe}$

$\mathit{UnsubscribeFromFilter} \mathrel{\widehat=} \mathit{channel.filter.Unsubscribe}$

$\mathit{RemoveProxyFromFilter} \mathrel{\widehat=} \mathit{channel.filter.Remove}$

I.8 *RTProxyPushSupplier*

The *RTProxyPushSupplier* extends the *ProxyPushSupplier* from the Channel pattern with interfaces for the consumers to subscribe and unsubscribe their interested subset of data. It also overrides the *DisconnectPushSupplier* operation so that the proxy and all the related subscription to the *Filter* are removed while the consumer disconnects itself from the proxy:

RTProxyPushSupplier

$\upharpoonright(\cdots, (\mathit{SubscribeInterests})$

$\mathit{ProxyPushSupplier}$

$\quad[\mathit{DisconnectPushSupplier}_{pps} / \mathit{DisconnectPushSupplier}];$

$consumerAdmin : \downarrow RTConsumerAdmin$

$SubscribeInterests \mathrel{\widehat=} [interests? : \mathbb{P} \downarrow DataInterest] \bullet$
$\quad consumberAdmin.SubscribeToFilter($
$\qquad proxy \leadsto self, interests \leadsto interests?)$

$UnsubscribeInterests \mathrel{\widehat=} [interests? : \mathbb{P} \downarrow DataInterests] \bullet$
$\quad consumberAdmin.UnsubscribeFromFilter($
$\qquad proxy \leadsto self, interests \leadsto interests?)$

$DisconnectPushSupplier \mathrel{\widehat=}$
$\quad DisconnectPushSupplier_{pps} \parallel$
$\quad consumberAdmin.RemoveProxyFromFilter(self)$

I.9 Other components

The other components of the push model of the Channel pattern, such as *PushSupplier*, *PushConsumer*, *ProxyPushConsumer* and *SupplierAdmin* are inherited by this pattern without any change. Similar extensions can also be easily done to the pull model of the Channel pattern to include timing and QoS control and data filtering. However as it has been argued at the beginning of the section, the push model is more suitable for real-time scheduling and synchronization tasks and hence it is used more often in our distributed multimedia system. The possible real-time extension to the pull model is omitted from this specification.

BACKGROUND MATERIAL J

Specifications of the Streaming Channel Pattern

This background material presents the formal specification of the components of the Streaming Channel pattern that is introduced in section 7.4 of chapter 7.

J.1 *Stream*

The incorporation of multimedia capability into an communication pattern requires a special type to carry the continuous data streams. Since multimedia streaming is supported by many existing protocols with QoS support, it is not necessary for us to model the technical details of streaming. Instead, these streams are modeled as of the class *Stream*. It wraps up the common attributes and functions of an active streaming flow:

Stream

$\upharpoonright(type, qos, dataSink, dataSource)$

$type : \mathbb{M}$
$qos : QoSProperties$
$dataSink : DataSink_{©}$
$dataSource : DataSource_{©}$

$RateConstrain \mathrel{\widehat=} [\, d? : Data \,] \bullet \textbf{let}\ rate == Get(qos, \text{“}rate\text{”}) \bullet$
$\qquad \textbf{if}\ rate \neq \mathsf{null}\ \textbf{then}\ [\, size(d?) \div (\tau' - \tau) \geqslant rate \,]$

$Transfer \mathrel{\widehat=} [\, d? : Data \,] \bullet RateConstrain \land$
$\qquad (dataSink.Fetch \mathbin{⨾} dataSink.Push)$

$$IdleTick \mathrel{\widehat=} [\lnot \mathrm{pre}(Transfer) \land \tau' = \tau + 1]$$
$$\textsc{Process} \mathrel{\widehat=} \mu\, S \bullet IdleTick \mathbin{[]} Transfer \mathbin{⨾} S$$

Every *Stream* has its media *type* (for example, a MIME type). Although the attribute *format* is not used in the specification here, it is included for other components to check the stream type. The *Stream* also has *qos* properties that provide the transportation that satisfies these QoS requirements, for example, the transportation *rate*. Unlike the Real-time Channel pattern with its dynamic support QoS control in every data delivery transaction, this pattern fixes the QoS requirement to each stream to decrease the data marshalling and unmarshalling overhead and stabilize the transportation. If there is a different QoS requirement on the communication, the communicating components must be disconnected from the stream and reconnected to a stream that satisfies the new requirement.

J.2 *DataSink, DataSource* and *DataSourceReceiver*

A *Stream* has two sides. At one side there is a *DataSink* object for data suppliers to push data into the sink, and at the other there is a *DataSource* for the consumers to fetch the data. In between, the active *Stream* object has a process that keeps getting data from the *DataSink* and putting it into the *DataSource*. Related QoS requirements must be satisfied in this process. As an example, the transportation must keep up with the *rate*:

$$size(d?) \div (\tau' - \tau) \geqslant rate$$

where the global function *size* returns the size of the data being transferred, and $\tau' - \tau$ is the time that the operation *Transfer* may take.

DataSink and *DataSource* are a subclass of *DataCache*:

DataCache

$$capacity : \mathbb{N}$$
$$\Delta$$
$$cache : \mathrm{seq}\, \mathbb{D}$$
$$\#cache \leqslant capacity$$

$$Fetch \mathrel{\widehat=} [\, d! : \mathbb{D} \mid cache = \langle d! \rangle \frown cache' \,]$$
$$Push \mathrel{\widehat=} [\, d? : \mathbb{D} \,]\ \mathbf{if}\ \#cache = capacity\ \mathbf{then}\ Fetch \setminus (d!) \mathbin{⨾}$$
$$cache := cache \frown \langle d? \rangle$$

The class *DataCache* is modeled as a queue that has a limited capacity to store data. Putting data into a full *DataCache* will cause dequeuing an element to give the space for the new element. The class *DataSink* simply hides the *Fetch* operation from other objects except the *Stream*:

DataSink

$\upharpoonright(Push, (Fetch))$
$DataCache$

Note that if the *Stream* does not fetch the data fast enough, the *DataSink* will be full and the data at the head of queue will be dropped to give space for the coming data. This requires the *Stream* must meet certain QoS requirements not to be slower than the data is pushed into the sink, otherwise data will be dropped even before it is transferred.

The class *DataSource* hides the inherited operations *Fetch* and *Push* from the other objects except the *Stream*. For other objects to get the data, *DataSource* applies an Observer pattern:

DataSource

$\upharpoonright(Subscribe, Unsubscribe, (Fetch, Push))$
$DataCache$

$receivers : \downarrow DataSourceReceiver$

$Subscribe \mathrel{\widehat{=}} [r? : \downarrow DataSourceReceiver] \bullet receivers := receivers \cup r?$
$Unsubscribe \mathrel{\widehat{=}} [r? : \downarrow DataSourceReceiver] \bullet receivers := receivers \setminus r?$

$PushToReceivers = Get \fatsemi \bigwedge r : receivers \bullet r.PushFromStream$
$IdleTick = IdleTick \mathrel{\widehat{=}} [\neg\ \mathrm{pre}(PushToReceivers) \wedge \tau' = \tau + 1]$
$\textsc{Process} = \mu\, DS \bullet IdleTick \mathbin{[]} PushToReceivers \fatsemi DS$

A *DataSource* object has an active process to fetch the data from its cache and push it to subscribed *DataSourceReceiver* objects. Again this process does not guarantee that every piece of the data will be pushed to the receivers. If the *DataSource* pushing process failed to keep up with the speed at which the data is filled into the cache, the data might be dropped out of the cache although it has already arrived in the cache for some time.

DataSourceReceivers may be added to or removed from the registry *receivers* with the operations *Subscribe* and *Unsubscribe*. A stream receiver should implement the *DataSourceReceiver* interface to receive the data forwarded from the Stream:

DataSourceReceiver

$PushFromStream \mathrel{\widehat{=}} [d? : \mathbb{D}]$

J.3 *StreamingChannel*

The *StreamingChannl* extends the class *Channel* from the Channel pattern with a *StreamAdmin* component. The *SupplierAdmin* and the *ConsumerAdmin* components are overridden with extended types that has a attribute variable to associate with the *StreamAdmin* component:

StreamingChannel

$\upharpoonright(\cdots)$

$Channel$

$supplierAdmin : \downarrow StreamSupplierAdmin_{\copyright}$

$consumerAdmin : \downarrow StreamConsumerAdmin_{\copyright}$

$streamAdmin : \downarrow StreamAdmin_{\copyright}$

$supplierAdmin.streamAdmin =$
$\quad consumerAdmin.streamAdmin = streamAdmin$

Note that the *StreamingChannel* also inherits the operations *ForSuppliers* and *ForConsumers* as interfaces for the suppliers and consumers to reach the *SupplierAdmin* and the *ConsumerAdmin* components.

J.4 *StreamAdmin*

The *StreamAdmin* provides the operation *CreateStream* creates *Stream* objects and manage them in a table of *streams*. The *streams* table is a partial bijective function that can be used to retrieve the reference to a stream with a given *id*?. Each *Stream* is created with a unique *id* and with its media type and QoS requirements specified. The *StreamAdmin* also provides the operation *DestroyStream* to destroy a stream and remove it from the *streams* table:

StreamAdmin

$\upharpoonright((CreateStream, DestroyStream, streams))$

Δ

$streams : String \rightarrowtail\!\!\!\!\!\twoheadrightarrow \downarrow Stream$

CreateStream

$id? : String$

$type? : \mathbb{M}$

$qos? : QoSProperties$

$stream : \downarrow Stream$

$id? \notin \mathrm{dom}(streams)$

$\mathrm{new}(stream, Stream)$

$stream.type = type? \wedge (stream.qos \Rightarrow qos?)$

$streams' = streams \cup \{id? \mapsto stream\}$

DestroyStream

$id? : String$

$id? \in \mathrm{dom}(streams) \land \mathrm{delete}(streams(id?))$
$streams' = \{id?\} \ndres streams$

J.5 *StreamSupplierAdmin*

The *StreamSupplierAdmin* extends the *SupplierAdmin* from the Channel pattern, with the operations for the suppliers to create a *Stream*, obtain and connect to a proxy consumer that is connected to a particular *Stream*. The operations for obtain and connect to push and pull proxy consumers are inherited and kept intact. The *StreamSupplierAdmin* has a reference to the *StreamAdmin* component of the channel, and a registry *streamConsumers* that keeps track of the proxy stream consumers:

StreamSupplierAdmin

$\upharpoonright(\cdots, CreateStream, ObtainStreamConsumer,$
$\quad DisconnectStreamConsumer)$
$SupplierAdmin[Destroy_{sa} / Destroy]$

$streamAdmin : \downarrow StreamAdmin$
$streamConsumers : \mathbb{P} \downarrow ProxyStreamConsumer_{©}$

The operation *CreateStream* promotes the same operation from the component *streamAdmin*. This operation is the only public interface of the channel for creating streams, which means only the stream suppliers may create a stream:

$CreateStream \mathrel{\widehat=} streamAdmin.CreateStream$

The operation *ObtainStreamConsumer* creates a proxy consumer that is connected to a stream, and adds it to the registry. The stream is required to have the same *id* as the input parameter *id*?, so for this operation to succeed, the target stream must have already been created:

ObtainStreamConsumer

$id? : String$
$streamConsumer! : \downarrow ProxyStreamConsumer$

$id? \in \mathrm{dom}(streamAdmin.streams)$
$\mathrm{new}(streamConsumer!, ProxyStreamConsumer)$
$streamConsumer!.\textsc{Init}$
$streamConsumer!.stream = streamAdmin.streams(id?)$
$streamConsumers' = streamConsumers \cup \{streamConsumer!\}$

Proxy consumers can be disconnected and removed from the registry by the operation *DisconnectStreamConsumer*:

$$\begin{array}{|l}
\textit{DisconnectStreamConsumer} \\
\hline
\textit{streamConsumer?} : \downarrow \textit{ProxyStreamConsumer} \\
\hline
\textit{streamConsumers}' = \textit{streamConsumers} \setminus \{\textit{streamConsumer?}\} \\
\hline
\end{array}$$

The operation *Destroy* from the super class is renamed to $\textit{Destroy}_0$ and included in the overriding operation. The overriding operation disconnects all connected push and pull proxy consumers, and proxy stream consumers as well:

$$\begin{array}{|l}
\textit{Destroy} \mathrel{\widehat=} \textit{Destroy}_{sa} \ \ \| \\
\qquad\qquad (\bigwedge sc : \textit{streamConsumers} \bullet sc.\textit{DisconnectStreamConsumer}) \\
\hline
\end{array}$$

J.6 *StreamConsumerAdmin*

The extension of *StreamConsumerAdmin* to *ConsumerAdmin* is similar to the extension of *StreamSupplierAdmin* to *SupplierAdmin*: the *StreamConsumerAdmin* also has a reference to the *StreamAdmin* component of the channel, and a registry *supplierConsumers* that keeps track of the proxy stream suppliers:

$$\begin{array}{|l}
\textit{StreamConsumerAdmin} \\
\hline
\upharpoonright(\cdots, \textit{ObtainStreamSupplier}) \\
\textit{ConsumerAdmin}[\textit{Destroy}_{ca} / \textit{Destroy}] \\
\\
\quad \begin{array}{|l}
\hline
\textit{streamAdmin} : \textit{StreamAdmin} \\
\textit{streamSuppliers} : \mathbb{P} \downarrow \textit{ProxyStreamSupplier}_{©} \\
\hline
\end{array}
\end{array}$$

The operation *ObtainStreamSupplier* is a two step process. It first invokes $\textit{ObtainSreamSupplier}_0$ to create a proxy supplier that is connected to a stream. The stream is required to have the same *id* as the input parameter *id*?. The created proxy supplier is also added to the registry for the management. Then the created proxy supplier, as a *DataSourceReceiver*, is subscribed to the *dataSource* of the stream for the incoming data:

$$\begin{array}{|l}
\textit{ObtainStreamSupplier} \mathrel{\widehat=} [\,\textit{id?} : \textit{String}\,] \bullet \\
\qquad (\textit{ObtainStreamSupplier}_0 \ \ \|_! \\
\qquad \textit{streamAdmin.streams}(\textit{id?}).\textit{dataSource.Subscribe}) \\
\\
\quad \begin{array}{|l}
\textit{ObtainStreamSupplier}_0 \\
\hline
\textit{id?} : \textit{String} \\
\textit{streamSupplier!} : \downarrow \textit{ProxyStreamSupplier} \\
\hline
\textit{id?} \in \operatorname{dom}(\textit{streamAdmin.streams}) \\
\text{new}(\textit{streamSupplier!}, \textit{ProxyStreamSupplier}) \\
\textit{streamSupplier!}.\textsc{Init} \\
\textit{streamSupplier!.stream} = \textit{streamAdmin.streams}(\textit{id?}) \\
\textit{streamSuppliers}' = \textit{streamSuppliers} \cup \{\textit{streamSupplier!}\} \\
\hline
\end{array}
\end{array}$$

The operation *DisconnectStreamConsumer* disconnects a proxy stream supplier and removes it from the registry. The proxy supplier is also also unsubscribed from the connected stream *dataSource*:

$$
\begin{array}{|l}
\mathit{DisconnectStreamSupplier} \mathrel{\widehat=} \\
\quad [\, \mathit{streamSupplier?} : \downarrow \mathit{ProxyStreamSupplier} \,] \bullet \\
\quad ([\, \mathit{streamSuppliers}' = \mathit{streamSuppliers} \setminus \{\mathit{streamSupplier?}\} \,] \parallel \\
\quad \mathit{streamSupplier?}.\mathit{stream}.\mathit{dataSource}.\mathit{Unsubscribe}(\mathit{streamSupplier?}))
\end{array}
$$

The operation *Destroy* from the super class is renamed to $\mathit{Destroy}_{ca}$ and included in the overriding operation. The overriding operation disconnects all connected push and pull proxy suppliers, and proxy stream suppliers as well:

$$
\begin{array}{|l}
\mathit{Destroy} \mathrel{\widehat=} \mathit{Destroy}_{ca} \parallel \\
\qquad\quad (\bigwedge \mathit{ss} : \mathit{streamSuppliers} \bullet \mathit{ss}.\mathit{DisconnectStreamSupplier}) \\
\hline
\end{array}
$$

J.7 *StreamSupplier*

A stream supplier supports *StreamSupplier* interface:

$$
\begin{array}{|l}
\hline
\mathit{StreamSupplier} \\
\hline
\mathit{DisconnectStreamSupplier} \mathrel{\widehat=} [\;] \\
\hline
\end{array}
$$

A concrete stream supplier should implement the operation *DisconnectStreamSupplier* to terminate the communication. It should release resources used at the supplier to support the communication. The *StreamSupplier* object reference should then be disposed. If a *StreamSupplier* object is connected to a *StreamConsumer* object, invoking the *DisconnectStreamSupplier* operation on the *StreamSupplier* object should also cause the implementation to call the *DisconnectStreamConsumer* operation on the *StreamConsumer* object.

J.8 *ProxyStreamSupplier*

The *ProxyStreamSupplier* implements both the *StreamSupplier* and the *DataSourceReceiver* interfaces. It also has a reference to its connected stream and a state variable *streamConsumer* to store the reference of the connected *StreamConsumer* object:

$$
\begin{array}{|l}
\hline
\mathit{ProxyStreamSupplier} \\
\hline
\upharpoonright(\mathit{stream}, \mathit{ConnectStreamConsumer}, \mathit{DisconnectStreamSupplier}, \\
\quad (\mathit{PushFromStream})) \\
\mathit{StreamSupplier}, \mathit{DataSourceReceiver}
\end{array}
$$

$stream : \downarrow Stream$
Δ
$streamConsumer : \downarrow StreamConsumer_{\circledcirc}$

Init

$streamConsumer = \mathsf{null}$

The operation *ConnectStreamConsumer* connects a *streamConsumer* object to the *ProxyStreamSupplier*, but only when it has not been connected to any other *streamConsumer* object:

ConnectStreamConsumer

$streamConsumer? : \downarrow StreamConsumer$

$streamConsumer = \mathsf{null} \land streamConsumer' = streamConsumer?$

The operation *DisconnectStreamSupplier* implements the corresponding abstract operation from its super class. If it is connected to a *StreamConsumer* object, the operation disconnects the *StreamConsumer* object and invokes the *DisconnectPullConsumer* on the *StreamConsumer* object. The *streamConsumer* attribute is then set to null, and the proxy supplier also disconnect itself from *consumerAdmin*:

```
$DisconnectStreamSupplier \mathrel{\widehat=}$
    $(\mathbf{if}\ streamConsumer \neq \mathsf{null}$
     $\mathbf{then}\ pushConsumer.DisconnectStreamConsumer) \parallel$
    $[\, streamConsumer' = \mathsf{null} \,] \parallel$
    $consumerAdmin.DisconnectStreamSupplier(self)$
```

The operation *PushFromStream* implements the *DataSourceReceiver* interface operation to receive the data from the stream and pushes it to connected *streamConsumer*:

$PushFromStream \mathrel{\widehat=} [d? : \mathbb{D}] \bullet \mathbf{if}\ streamConsumer \neq \mathsf{null}$
$\quad \mathbf{then}\ streamConsumer.Push(d?)$

J.9 *StreamConsumer*

A stream consumer provides the operations defined in the class *PushConsumer*:

StreamConsumer

$Push \mathrel{\widehat=} [\, d? : \mathbb{D} \,]$
$DisconnectStreamConsumer \mathrel{\widehat=} [\ ]$

A supplier communicates data to the consumer by invoking the operation *Push* which takes the data $d?$ as input. The operation *DisconnectStreamConsumer* terminates the communication. It should release resources used at the consumer to support the communication. The *StreamConsumer* object reference should then be disposed. If a *StreamConsumer* object is connected to a *StreamSupplier* object, invoking the *DisconnectStreamConsumer* operation on the *StreamConsumer* object should also cause the implementation to call the *DisconnectStreamConsumer* operation on the *StreamSupplier* object.

J.10 *ProxyStreamConsumer*

The class *ProxyStreamConsumer* implements the abstract interfaces defined in *StreamConsumer*. It also defines the operations for connecting push suppliers to a channel. It also has a reference to its connected stream and a state variable *streamSupplier* to store the reference of the connected *StreamSupplier* object:

$$
\begin{array}{|l}
\textit{ProxyStreamConsumer} \\
\hline
\upharpoonright(\textit{stream}, \textit{ConnectStreamSupplier}, \textit{DisconnectStreamConsumer}, \textit{Push}) \\
\textit{StreamConsumer} \\
\\
\quad \begin{array}{|l}
\hline
\textit{stream} : \downarrow \textit{Stream} \\
\Delta \\
\textit{streamSupplier} : \downarrow \textit{StreamSupplier}_{\circledcirc} \\
\hline
\end{array} \\
\\
\quad \begin{array}{|l}
\textsc{Init} \\
\hline
\textit{streamSupplier} = \mathsf{null} \\
\hline
\end{array}
\end{array}
$$

The operation *ConnectStreamSupplier* connects a *StreamSupplier* object to the *ProxyStreamConsumer*, but only when it has not been connected to any other *StreamSupplier* object:

$$
\begin{array}{|l}
\quad \begin{array}{|l}
\textit{ConnectStreamSupplier} \\
\hline
\textit{streamSupplier}? : \downarrow \textit{StreamSupplier} \\
\hline
\textit{streamSupplier} = \mathsf{null} \land \textit{streamSupplier}' = \textit{streamSupplier}? \\
\hline
\end{array}
\end{array}
$$

The operation *ConnectStreamSupplier* implements the semantics of the corresponding abstract operation from its super class. If it is connected to a *StreamSupplier* object, the operation disconnects the *StreamSupplier* object and invokes the *DisconnectStreamSupplier* on the *StreamSupplier* object. The operation also disconnects this proxy consumer from the *SupplierAdmin* component:

$$
\begin{array}{|l}
\textit{DisconnectStreamConsumer} \mathrel{\widehat=} \\
\qquad (\mathbf{if}\ \textit{streamSupplier} \neq \mathsf{null} \\
\qquad\ \ \mathbf{then}\ \textit{streamSupplier}.\textit{DisconnectStreamSupplier}) \parallel \\
\qquad [\textit{streamSupplier}' = \mathsf{null}] \parallel \\
\qquad \textit{supplierAdmin}.\textit{DisconnectStreamConsumer}(\textit{self})
\end{array}
$$

The operation *Push* implements the abstract operation from its super class to receive the data from the *StreamSupplier* objects and push the data directly into the connected stream:

$$Push \mathrel{\widehat{=}} [\, d? : \mathbb{D} \,] \bullet stream.dataSink.Push(d?)$$

BACKGROUND MATERIAL K

Specifications of the Distributed MVC Pattern

This background material presents the formal specification of the components of the Distributed MVC pattern that is introduced in section 8.2 of chapter 8.

K.1 *Model*

The *Model* component encapsulates the appropriate data and serves a set of operations on the data. It processes the data according to the input event that controllers send on behalf of the users, and notify the depending components (views and controllers) about the update if there is any. It also serves its data or certain portions of the data according to the query requests when the depending components need to refresh its presentation. The relation between depending components and the model are often implemented using the Observer pattern (Bachmann et al., 2000). The Observer pattern uses direct invocation of the operations of the model component to send the input and query and direct invocation of the operations of the depending components to notify the changes and send back the query results. However direct invocations are not possible in a distributed environment. Although it can be implemented using RMI or RPC mechanisms as if the invocations are local calls, it is then prone to synchronous communication: the calling process is blocked while the call is processed by the remote serving component. To enable asynchronous communication between the model and the depending components, here the communication is modeled using the push-style of the Channel pattern from the previous chapter. In principle, other styles of the Channel pattern can also be used, but as it is argued in section 7.3.3 on page 89, the push style is more appropriate in real-time missions for scheduling and synchronization tasks, hence let's again focus on this model and present the solution based on the push model as an example.

Instead of providing the operations for the depending components to make direct invocations, the *Model* component below only exposes its input and output ports that can be connected to the channels and further to the input and output ports of the depending components, leaving the definitions of input events, state change notifications, queries and query results open:

$$\mathit{Model}[\mathit{Input}, \mathit{Notification}, \mathit{Query}, \mathit{QueryResult}]$$

$$\upharpoonright(\mathit{inouts})$$

$$\mathit{inouts} : \mathit{MPushConsumerImpl}_{©} ⤔ \mathit{PushSupplierImpl}_{©} \quad \text{[I/O ports]}$$

$$\forall\, \mathit{in} \mapsto \mathit{out} : \mathit{inouts} \bullet \mathit{in}.m = \mathit{self}$$

An “M” is added in front of *PushConsumer* and an “Impl” is attached after to indicate an **impl**ementation of *PushConsumer* that has a reference *m* to the **M**odel. The specification of *MPushConsumerImpl* will appear soon on the facing page.

The input and output ports *inouts* (*PushConsumer* and *PushSupplier* components) are organized in pairs to provide bidirectional communication between the *Model* component and the depending components, and to relate the output channel to the input channel if necessary. The input ports not only implement the *PushConsumer* interface, but also have a reference to the containing *Model* component so that they can invoke appropriate operations of the *Model* component when data is received. The output ports simply provide a *Push* operation to delegate the output from the Model component to connected channels.

The *Model* component has an operation to handle the input events received from its input ports. It is modeled as a two step process: the operation *ProcessInput* processes the input and if necessary, and creates state change notification; if the depending components indeed need to be notified, the operation *HandleInput* pushes the notifications to the outport ports:

$$\mathit{ProcessInput}$$

$$i? : \downarrow \mathit{Input}$$
$$n! : \downarrow \mathit{Notification}_{\circledcirc}$$

Process the input event;
Create state change notification.

$$\begin{aligned}\mathit{HandleInput} \mathrel{\widehat{=}} {} & \mathit{ProcessInput} \parallel \\ & ([\, n? : \downarrow \mathit{Notification}_{\circledcirc} \,] \bullet \\ & \quad \textbf{if}\, n? \neq \mathsf{null}\, \textbf{then} \bigwedge \mathit{in} \mapsto \mathit{out} : \mathit{inouts} \bullet \mathit{out}.\mathit{Push}(n?))\end{aligned}$$

The operation *ProcessInput* here is only a skeleton with only the input and output interfaces specified. The concrete logic of this operation is intended to be implemented or overridden by the applications. The same two step technique is used in other specifications here: the application logic is dealt with as an abstract operation and concentrate only on event and data handling. Immediately there is another example:

ProcessQuery

$q? : \downarrow Query$
$r! : \downarrow QueryResult$

⟨Process the query $q?$ and generate the query result $r!$⟩
$session(r!) = session(q?)$

$HandleQuery \mathrel{\widehat{=}} [from? : MPushConsumerImpl;\ q? : \downarrow Query] \bullet$
$\quad (ProcessQuery(q?) \parallel$
$\quad\ [r? : \downarrow QueryResult] \bullet inouts(from?).Push(r?))$

The *Model* processes the incoming queries and generates the query results with the operation *ProcessQuery* (the first step, an abstract operation without completely specifying the concrete logic), and deliver the query results to the output port that is paired with the input port where the query comes from (the second step, with concrete semantics).

Since the communication between the querying component and the *Model* is asynchronous, it is often necessary for the querying component to know which query the result is related to. In most of the distributed systems, this is managed with sessions. The query result is marked with the same session identification as the query so that a dialog can be established between the involved parties. Without loosing its generality, these session identifications are modeled as strings. Let's assume that there is a global function *session* that may get the session identifications from *Query* objects and *QueryResult* objects:

$$session : \downarrow Query \cup \downarrow QueryResult \rightarrow String$$

With the function *session* defined, the operation *ProcessQuery* must identify the query result $r!$ with the same session with the input query $q?$, such that $session(r!) = session(q?)$.

The input and outport ports of the *Model* implement the classes *PushConsumer* and *PushSupplier* from the Channel pattern:

MPushConsumerImpl[*Input*, *Query*]

$PushConsumer[Push_{pc} / Push]$

$m : \downarrow Model$

$Push \mathrel{\widehat{=}} [d? : \mathbb{O}] \bullet$
$\quad$ **if** $d? \in \downarrow Input$ **then** $m.HandleInput(d?)$
$\quad$ **else if** $d? \in \downarrow Query$ **then** $m.HandleQuery(from \leadsto self, q \leadsto d?)$

The input ports (*MPushConsumerImpl* objects) are push consumers. Each port implements the *Push* interface of *PushConsumer* to receive the data pushed from the connected channel. Depending on the type of the incoming data, it invokes the operation *HandleInput* of the related Model component to hand over the input events, or the operation *HandleQuery* of the Model to identify the port itself and forward the received query $q?$.

The output ports are push suppliers, modeled as the type *PushSupplierImpl*. A *PushSupplierImpl* object is to be connected to a proxy push consumer from the channel. The output behavior of the *Model* is modeled as an operation of invoking the *Push* operation of the output port and this *Push* operation directly promotes the *Push* operation of the connected proxy consumer:

PushSupplierImpl

$PushSupplier$

$proxy : \downarrow ProxyPushConsumer$

$Push \mathrel{\widehat=} proxy.Push$

K.2 *ModelDependent*

The *view* and the *controller* are *ModelDependent* components. They observe the state changes of a model. If the Real-time Channel pattern is used, they may also subscribe their interests of data so that only interested state changes of the model will be delivered to them. Otherwise, the notified *ModelDependent* may still check whether the update is interested upon receiving. The notification may have direct influence on the interface, or if necessary, a query will be sent back to model to retrieve the updated state. When the query result is received, The *ModelDependent* prepares the data for platform dependent presentation, for example, by adding local color schemes and layout arrangements to a graphical presentation. It then presents the prepared data to the environment.

Every *ModelDependent* has an input port and an output port connected through channels to the *Model* component:

ModelDependent[*Notification*, *Query*, *QueryResult*]

$\upharpoonright(in, out)$

$in : MDPushConsumerImpl_{©}$ [receiving data from model]
$out : PushSupplierImpl_{©}$ [sending data to model]

$in.md = self$

Every *ModelDependent* component, either a *View* component or a *Control* component, has a function to present the data or render its input controllers to the "real physical world". It is modeled as an operation as follows (see also section F.8 on page 288):

$$Present \mathrel{\widehat=} \left[d? : \mathbb{D}_{\circledcirc};\ r : \mathbb{R};\ pr : \mathbb{D} \rightarrow \mathbb{R} \;\middle|\; d? \neq \mathsf{null} \Rightarrow r = pr(d?) \right]$$

It may directly render the data to the “real physical world”, or send the data to the rendering engine in the underlying platform. Let’s leave it to the implementation.

The operation *ProcessNotification* processes the received notification. If it has direct impact on the interface, the operation produces the data $d!$ for presentation. If more data is needed for updating the interface, a query $q!$ is generated for sending back to the *Model* component. If the notification $n?$ is not in the interests at all, both the outputs $d!$ and $q!$ will be set to null. The operation *HandleNotification* takes the output of *ProcessNotification*, updates the interface by invoking the operation *Present* if the prepared d is not null, and pushes the query to the output port if necessary:

ProcessNotification

$n? : \downarrow Notification$
$d! : \mathbb{D}_{\circledcirc}$
$q! : \downarrow Query_{\circledcirc}$

If the notification has direct influence on the user interface, output $d!$ for presentation;
If it requires more information from the model to update the user interface, create the query $q!$.

$$HandleNotification \mathrel{\widehat=} ProcessNotification \parallel (Present \parallel [q? : \downarrow Query_{\circledcirc}] \bullet \mathbf{if}\ q? \neq \mathsf{null}\ \mathbf{then}\ out.Push(q?))$$

The returned query result is processed by the operation *ProcessQueryResult*, then presented to the interface by the operation *HandleQueryResult*:

ProcessQueryResult

$r? : \downarrow QueryResult$
$d! : \mathbb{D}_{\circledcirc}$

Process the received query result, and prepare the data $d!$ for presentation

$$HandleQueryResult \mathrel{\widehat=} ProcessQueryResult \parallel Present$$

The input port of the *ModelDependent* component is of the type *MDPushConsumerImpl*. It implements the *Push* interface of *PushConsumer* to receive the data from a channel. Depending on the type of the received data, it invokes the corresponding operation of the related *ModelDependent* component to handle the notification or the query result:

MDPushConsumerImpl[*Notification*, *QueryResult*]

$PushConsumer[Push_{pc} / Push]$

$md : \downarrow ModelDependent$

$Push \mathrel{\widehat=} [\, d? : \mathbb{O} \,] \bullet$
$\quad \textbf{if } d? \in \downarrow Notification \textbf{ then } md.HandleNotification(d?)$
$\quad \textbf{else if } d? \in \downarrow QueryResult \textbf{ then } md.HandleQueryResult(d?)$

K.3 *View*

The *View* extends *ModelDependent* with an additional operation to handle the input event directly sent from the controller, which allows alternation of the view without updating the model (e.g. zooming in and out a graphical presentation, increasing or decreasing the volume of an audio presentation):

$View[Input, Notification, Query, QueryResult]$

$\upharpoonright(\cdots, (HandleInput))$
$ModelDependent$

$ProcessInput$

$i? : \downarrow Input$
$d! : \mathbb{D}_{\circledcirc}$

Process the input event;
Prepare data for presentation.

$HandleInput \mathrel{\widehat=} ProcessInput \parallel Present$

K.4 *Controller*

The *Controller* also inherits the observing, querying and rendering behavior of *ModelDependent*, because in many cases, the controller needs to render its interface of its own, according to the state of the associated model, for example a menu or a group of buttons which items may be enabled or disabled depending on the state of the model:

$Controller[Input, Notification, Query, QueryResult]$

$\upharpoonright(\cdots, (v))$
$ModelDependent$

It adds a state variable v to refer to the associated v component:

$v : \downarrow View$ [the coupled view component]

It processes the raw input and transforms it to input events. The input events can be an event $i_v!$ for the view to update its presentation directly, and/or an input $i_m!$ for the model to update its state. The raw input may also have direct influence on how the control component is rendered on the user interface, the data is then prepared for rendering:

ProcessInputData

$d? : \mathbb{D}$
$i_m!, i_v! : \downarrow Input_{\circledcirc}$
$d! : \mathbb{D}_{\circledcirc}$

Process the raw input;
Create the input event $i_m!$ for the model;
Create the input event $i_v!$ for the view component;
If the input has direct influence on the interface, prepare the data $d!$ for presentation.

$$\begin{aligned}
&HandleInputData \mathrel{\widehat=} ProcessInputData \parallel \\
&\quad ([\, i_m?, i_v? : \downarrow Input_{\circledcirc};\ d? : \mathbb{D}_{\circledcirc} \,] \bullet \\
&\quad\ ((\mathbf{if}\, i_m? \neq \mathsf{null}\, \mathbf{then}\, out.Push(i_m?)) \parallel \\
&\quad\ \ (\mathbf{if}\, i_v? \neq \mathsf{null}\, \mathbf{then}\, v.HandleInput(i_v?)) \parallel \\
&\quad\ \ Present(d?)))
\end{aligned}$$

The operation *HandleInputData* is an interfacing operation for the source of the input to submit the raw input. The source can be a sensor in the environment, the event mechanism of the underlying platform, or the *Controller* itself that has an active process to pull the input directly from the "real physical world":

ActiveController[*Input*, *Notification*, *Query*, *QueryResult*]

$\upharpoonright(\cdots)$
Controller

$$\begin{aligned}
&PullInput \mathrel{\widehat=} [\, r : \mathbb{R};\ pull : \mathbb{R} \to \mathbb{D}_{\circledcirc};\ d! : \mathbb{D} \mid \\
&\qquad\qquad \mathbf{let}\, d == pull(r) \bullet d \neq \mathsf{null} \land d! = d \,] \\
&IdleTick \mathrel{\widehat=} [\, \neg\ \mathrm{pre}(GetInputData) \land \tau' = \tau + 1 \,] \\
&\textsc{Process} \mathrel{\widehat=} \mu\, AC \bullet (IdleTick \mathrel{[]} \\
&\qquad\qquad (PullInput \parallel HandleInputData)) \fatsemi AC
\end{aligned}$$

The operation *PullInput* is the reversed operation of *Present*: the controller has a function to *pull* the data from the "real physical world". The active process keeps pulling the data and once the data is available, invokes the operation *HandleInputData* by itself.

BACKGROUND MATERIAL **L**

Specifications of the Distributed PAC Pattern

This background material presents the formal specification of the components of the Distributed PAC pattern that is introduced in section 8.3 of chapter 8.

L.1 *Abstraction*

To get an insight how the PAC agents work together, let's first have a look at their *Abstraction* components. With no direct connection with the *Presentation* component, the *Abstraction* component has a local reference to the *Control* component and provides two interfacing operations *HandleInput* and *HandleQuery* for the *Control* component to send the *Input* events and *Query* messages:

```
── Abstraction[Input, Notification, Query, QueryResult] ──────────
│ ↾(c, HandleInput, HandleQuery)
│
│  ┌──────────────────────────────────────────────
│  │ c : ↓Control
│  └──────────────────────────────────────────────
```

Once an *Input* event is received, the *Abstraction* component processes the input event and decides what to do with it. It may result in internal state changes that are in the interests of other components. *Notifications* are then created and sent to the associated *Control* by invoking its *HandleNotification* operation:

```
│  ── ProcessInput ──────────────────────────────
│  │ i? : ↓Input
│  │ n! : ↓Notification©
│  ├──────────────
```

```
Process the input event;
Create state change notification if there is any.
```

$HandleInput \mathrel{\widehat{=}} ProcessInput \parallel$
  $[\, n? : \downarrow Notification_{\circledcirc} \,] \bullet$
    if $n? \neq$ null **then** $c.HandleNotification(n?)$

Note that the notification is sent by invoking the corresponding operation of the *Control* component instead of directly returning the notification as an output of the operation *HandleInput*. There are two reasons behind this design decision. Firstly, the *Abstraction* component can be active and the internal state may change without being triggered by the input events. The *Control* component must provide an interfacing operation for the *Abstraction* to notify the change, otherwise the *Control* must constantly pull the change and this can be processing intensive. Hence the *Control* should take a passive role. Secondly, the *Abstraction* is not the only component sending the change notifications. The upper-level agents might also push notifications down to the *Control* component, it must have an interface *HandleNotification* to receive the notification from the channel. It is better to have the same mechanism for handling the notifications that may come from different places.

Upon receiving a *Query* for its data, the *Abstraction* component creates the output and associates the query result with the query by assigning the same session identification to the result:

ProcessQuery

$q? : \downarrow Query$
$r! : \downarrow QueryResult$

‹Process the query $q?$ and generate the query result $r!$›
$session(r!) = session(q?)$

$HandleQuery \mathrel{\widehat{=}} ProcessQuery \parallel c.HandleQueryResult$

The result is returned by a reversed invocation *c.HandleQueryResult* instead of returning the results to the *Control* component, as it was done with the notifications.

Comparing to the *Model* component in MVC, the responsibility of the *Abstraction* component in PAC is similar. It updates its internal state according to the input events and notifies other components with the updates if necessary. It also responds to data queries with the data concerned. The difference is that the *Abstraction* component does not connect to the user interface directly – there is no more than one component it needs communicate with, that is, the control. Another difference is that the *Abstraction* and the *Control* components are locally coupled and the communication in between is done through synchronous operation invocation. No channel is needed and the *Abstraction* does not need to maintain the channel connections as the *Model* does. Communications with the *Presenation* component, and with other PAC agents, are mediated and managed by the dedicated *Control* component. The *Abstraction* of PAC is purely focused on the core functions and data repository, whereas the *Model* component also has to take care of communication if multiple *Models* are needed in a distributed environment.

L.2 *Presentation*

As it is found in many implementations of the MVC pattern, separating the input and the output at the user interface is not only difficult, but also confusing. The PAC pattern actually combines the *View* and the *Controller* from MVC as one integrated *Presentation* component. It not only takes the raw input from the environment or the underlying platform with operation *HandleInputData* as the MVC *Controller* does, but also updates the internal state to the user interface with operations *HandleNotification* and *HandleQueryResult* as the MVC *View* does:

$$\begin{array}{|l}
\textit{Presentation}[\textit{Input}, \textit{Notification}, \textit{Query}, \textit{QueryResult}] \\
\upharpoonright(\textit{HandleInputData}, \textit{HandleNotification}, \textit{HandleQueryResult}) \\
\hline
c : \downarrow \textit{Control} \\
\hline
\end{array}$$

The difference between the *Presentation* and the MVC view-controller pair is that the *Presentation* component here does not talk to the data and logic core directly, but a *Control* component *c* in between, so that the *Presentation* component focuses only on the user interface, and leaves the issues of synchronous or asynchronous communication with local or remote components to locally a coupled *Control* component.

As a user interface component, the *Presentation* component has a function to *Present* data to the “real physical world”:

$$\textit{Present} \mathrel{\widehat{=}} [\, d? : \mathbb{D}_{\circledcirc};\ r : \mathbb{R};\ pr : \mathbb{D} \rightarrow \mathbb{R} \mid d? \neq \mathsf{null} \Rightarrow r = pr(d?) \,]$$

The *Presentation* component processes the received notification, update the user interfaces if it has direct impact on the interface and create queries if more data is needed for refreshing the interface, as the user interface components do in MVC. However without connection with any channel, the notification is received directly from invocations made by the *Control* *c* and the query is sent directly to *c* by invoking the operation *c.HandleQuery*:

$$\begin{array}{|l}
\textit{ProcessNotification} \\
\hline
n? : \downarrow \textit{Notification} \\
d! : \mathbb{D}_{\circledcirc} \\
q! : \downarrow \textit{Query}_{\circledcirc} \\
\hline
\text{If the notification has direct influence on the user interface, output } d! \\
\text{for presentation;} \\
\text{If it requires more information from the abstract component to update} \\
\text{the user interface, create the query } q!. \\
\hline
\end{array}$$

$$\begin{aligned}
&\mathit{HandleNotification} \mathrel{\widehat=} \mathit{ProcessNotification} \parallel \\
&\quad (\mathit{Present} \parallel [q? : \downarrow \mathit{Query}_{\circledcirc}] \bullet \\
&\qquad\qquad \textbf{if}\, q? \neq \mathsf{null} \\
&\qquad\qquad \textbf{then}\, c.\mathit{HandleQuery}(\mathit{from} \leadsto \mathit{self}, q \leadsto q?))
\end{aligned}$$

The operation *HandleQueryResult* does exactly the same as it does in MVC user interface components – updating the user interface with the query result, but the query result is received from the *Control* instead of an asynchronous channel:

ProcessQueryResult

$r? : \downarrow \mathit{QueryResult}$
$d! : \mathbb{D}_{\circledcirc}$

Process the received query result, and prepare the data $d!$ for presentation

$$\mathit{HandleQueryResult} \mathrel{\widehat=} \mathit{ProcessQueryResult} \parallel \mathit{Present}$$

As said, the *Presentation* component also gets the user input from the environment or the underlying platform as the MVC *Controller* does. Again the input is sent directly to the mediating *Control*, instead of a channel:

ProcessInputData

$d? : \mathbb{D}$
$i! : \downarrow \mathit{Input}_{\circledcirc}$
$d! : \mathbb{D}_{\circledcirc}$

Process the raw input;
Create the input event $i!$;
If the raw input has direct influence on the user interface, prepare the data for presentation.

$$\begin{aligned}
&\mathit{HandleInputData} \mathrel{\widehat=} \mathit{ProcessInputData} \parallel \\
&\quad ([i? : \downarrow \mathit{Input}_{\circledcirc};\ d? : \mathbb{D}_{\circledcirc}] \bullet \\
&\qquad ((\textbf{if}\, i? \neq \mathsf{null}\, \textbf{then}\, c.\mathit{HandleInput}(i?)) \parallel \mathit{Present}(d?)))
\end{aligned}$$

A *Presentation* component can also be implemented with an active process to *pull* input by itself from the “real physical world” $\mathbb{R}$ with the operation *PullInput*, instead of passively waiting for other components to *push* the raw input in:

ActivePresentation[*Input*, *Notification*, *Query*, *QueryResult*]

$\upharpoonleft(\cdots)$
Presentation

$$\begin{aligned}
&\mathit{PullInput} \mathrel{\widehat=} [r : \mathbb{R};\ \mathit{pull} : \mathbb{R} \to \mathbb{D}_{\circledcirc};\ d! : \mathbb{D} \mid \\
&\qquad\qquad \textbf{let}\, d == \mathit{pull}(r) \bullet d \neq \mathsf{null} \land d! = d]
\end{aligned}$$

$$\mathit{IdleTick} \mathrel{\widehat=} [\neg\ \mathrm{pre}(\mathit{PullInput}) \land \tau' = \tau + 1]$$

$$\begin{array}{l}\textsc{Process} \mathrel{\widehat=} \mu\, AP \bullet (\mathit{IdleTick} \mathrel{[]} \\ \qquad\qquad (\mathit{GetInputData} \parallel \mathit{HandleInputData}))\ ⨾\ AP\end{array}$$

L.3 *Control*

The *Control* component sits between the *Presentation* component and the *Abstraction* component. It forwards the *Input* events and *Query* messages from *Presentation* to *Abstraction*, and forwards the change *Notifications* and *QueryResults* from *Abstraction* to *Presentation*. "Forward" might be a wrong word here, because during this process the *Control* also has to decide whether the received event or data, or part of it, should be forwarded to its locally coupled components, or to other connected PAC agents in the hierarchy, or both. The event or data might also be transformed depending on the destination. The *Control* is not just a broker to simply hand over whatever it receives, but also a forger to reshape the events and the data for the destination, and a manager to make decisions on the directions of the data flow (figure L.1).

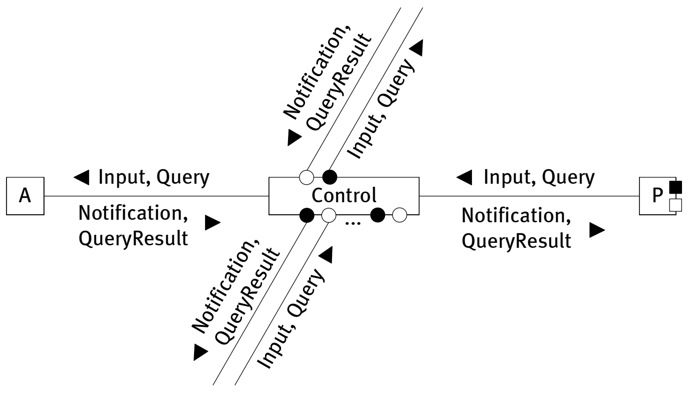

Notification, QueryResult ▶
Input, Query ◀
A
◀ Input, Query
Notification, QueryResult ▶
Control
...
◀ Input, Query
Notification, QueryResult ▶
P
Notification, QueryResult ▶
Input, Query ◀

Figure L.1: Data flow in a PAC agent

The *Control* component has direct references to its associated *Abstraction* a and *Presentation* p, a pair of input and output ports (in_{up}, out_{up}) for connecting with the upper-level component, a collection of input and output port pairs for connecting with lower-level components. Input and output ports (*inouts*) are modeled as push consumers and suppliers as they are in the distributed MVC pattern:

$$\begin{array}{l}\mathit{Control}[\mathit{Input}, \mathit{Notification}, \mathit{Query}, \mathit{QueryResult}] \\ \upharpoonright(in_{up}, out_{up}, inouts_{down} \\ \quad \mathit{HandleInput}, \mathit{HandleQuery}, \mathit{HandleNotification}, \mathit{HandleQueryResult}) \\ \hline a : \downarrow \mathit{Abstraction} \\ p : \downarrow \mathit{Presentation}\end{array}$$

$in_{up} : CPushConsumerImpl_{\copyright}$ [From the up-layer component]
$out_{up} : PushSupplierImpl_{\copyright}$ [To the up-layer component]
$inouts_{down} : CPushConsumerImpl_{\copyright} \pinj PushSupplierImpl_{\copyright}$
[I/O ports for down-layer components]
Δ
$sessions : Dictionary$

$in_{up}.c = self$
$\forall\, in \mapsto out : inouts_{down} \bullet in.c = self$

The *Control* also maintains a dynamic registry of query *sessions*, modeled as a *Dictionary* that keeps session identifications as keys and querying components as values. Since the query results may come back later from an asynchronous source, it is necessary to keep such a registry in order to return the query results back to the component that queried with the same session identification.

The *Control* component processes the input event received from both the *Presentation* component and the lower-level PAC agents. It decides whether to reform and forward it to the *Abstraction* component, or to the upper-level agent, or both:

ProcessInput
$i? : \downarrow Input$
$i_a!, i_{up}! : \downarrow Input_{\circledcirc}$

Generate the input event $i_a!$ for the abstract component;
Generate the input event $i_{up}!$ for upper-level component.

$$
\begin{aligned}
&HandleInput \mathrel{\widehat=} ProcessInput \parallel\\
&\quad ([\, i_a?, i_{up} : \downarrow Input_{\circledcirc} \,] \bullet\\
&\quad\ (\mathbf{if}\ i_a? \neq \mathsf{null}\ \mathbf{then}\ a.HandleInput(i_a?)) \parallel\\
&\quad\ (\mathbf{if}\ i_{up}? \neq \mathsf{null}\ \mathbf{then}\ out_{up}.Push(i_{up}?)))
\end{aligned}
$$

The *Control* component also has operations to process the change notifications from both the associated *Abstraction* component and the upper-level PAC agent. It has to make decisions whether the notification should be transformed, and whether the notification should be sent to *Presentation* and/or to the lower-level PAC agents:

ProcessNotification
$n? : \downarrow Notification$
$n_p!, n_{down}! : \downarrow Notification_{\circledcirc}$

Generate the notification $n_p!$ for the presentation component;
Generate the message $n_{down}!$ for down-layer components.

$$
\begin{aligned}
&HandleNotification \mathrel{\widehat=} ProcessNotification \parallel\\
&\quad ([\, n_p?, n_{down}? : \downarrow Notification_{\circledcirc} \,] \bullet\\
&\quad\ (\mathbf{if}\ n_p? \neq \mathsf{null}\ \mathbf{then}\ p.HandleNotification(n_p?)) \parallel\\
&\quad\ (\mathbf{if}\ n_{down}? \neq \mathsf{null}\\
&\quad\ \mathbf{then} \textstyle\bigwedge in \mapsto out : inouts_{down} \bullet out.Push(n_{down}?)))
\end{aligned}
$$

The *Control* component may receive queries from both the *Presentation* component and the lower-level PAC agents. It has to decide whether a received query should be sent to the associated *Abstraction* component or to the up level, or both. Since the query result might be returned asynchronously, the querying component is registered in the registry with the session identification as the key:

```
ProcessQuery
  $from? : \mathbb{O}$
  $q? : \downarrow Query$
  $q_a!, q_{up}! : \downarrow Query_{\circledcirc}$
  ----------------------------------------
  ⟨Generate the query $q_a!$ for the abstraction component⟩
  ⟨Generate the query $q_{up}!$ for the up-layer component⟩
  $q_a! \neq \mathsf{null} \Rightarrow sessions' = Put(sessions, session(q_a!) \mapsto from?)$
  $q_{up}! \neq \mathsf{null} \Rightarrow sessions' = Put(sessions, session(q_{up}!) \mapsto from?)$
```

```
$HandleQuery \mathrel{\widehat{=}} ProcessQuery \parallel$
    $([\, q_a?, q_{up}? : \downarrow Query_{\circledcirc} \,] \bullet$
     $(\mathbf{if}\ q_a? \neq \mathsf{null}\ \mathbf{then}\ a.HandleQuery(q_a?)) \parallel$
     $(\mathbf{if}\ q_{up}? \neq \mathsf{null}\ \mathbf{then}\ out_{up}.Push(q_{up}?)))$
```

Upon receiving a query result from the *Abstraction* or the upper-level PAC agent, the operation *HandleQueryResult* first looks up the registry to find the component that queried with the same session identification, then forwards the result to this component: if it was the *Presentation* that queried, its *HandleQueryResult* operation is invoked to pass over the result; or if the query was from a input port connected to a lower-level PAC agent, the result is then pushed into the paired output port to send the result over. At the same time, the completed session is removed from the registry:

```
$HandleQueryResult \mathrel{\widehat{=}} [\, r? : \downarrow QueryResult \,] \bullet$
    $\mathbf{let}\ s == session(r?) \bullet \mathbf{let}\ from == Get(sessions, s) \bullet$
      $\mathbf{if}\ from \neq \mathsf{null}$
      $\mathbf{then}\ (sessions := sessions \setminus \{s \mapsto from\} \parallel$
          $\mathbf{if}\ from = p\ \mathbf{then}\ p.HandleQueryResult(r?)$
          $\mathbf{else\ if}\ from \in \mathrm{dom}(inouts)\ \mathbf{then}\ inouts_{down}(from).Push(r?))$
```

The output ports are push suppliers of the type *PushSupplierImpl* as they are in the distributed MVC pattern (see definition on page 352). The input ports are push consumers of the type *CPushConsumerImpl*:

```
CPushConsumerImpl[Input, Notification, Query, QueryResult]
  $PushConsumer[Push_{pc} / Push]$

  ----------------------------------------
  $c : \downarrow Control$
  ----------------------------------------
```

```
$Push \mathrel{\widehat=} [\, d? : \mathbb{O} \,] \bullet$
    **if** $d? \in \downarrow Input$
    **then** $c.HandleInput(d?)$
    **else if** $d? \in \downarrow Notification$
        **then** $c.HandleNotification(d?)$
        **else if** $d? \in \downarrow Query$ **then** $c.HandleQuery(from \leadsto self, q \leadsto d?)$
            **else if** $d? \in \downarrow QueryResult$ **then** $c.HandleQueryResult(d?)$
```

It implements the *Push* operation of *PushConsumer* to receive data from the connected channel. Depending on the type of the received data, it invokes corresponding operations of the associated *Control* component.

L.4 *PAC* agent

Modeling a PAC agent is straightforward. A PAC agent is a composition of an *Abstraction* component, a *Control* component and a *Presentation* interface with all attributes and operations hidden, but only with the input and output ports from the *Control* component exposed for channel connections (figure L.2):

PAC

$\upharpoonright(in_{up}, out_{up}, inouts_{down})$

$p : \downarrow Presentation_{©}$
$a : \downarrow Abstraction_{©}$
$c : \downarrow Control_{©}$
$in_{up} : CPushConsumerImpl$
$out_{up} : PushSupplierImpl$
$inouts_{down} : CPushConsumerImpl \leftrightarrow PushSupplierImpl$

$p.c = a.c = c \land c.a = a \land c.p = p$
$in_{up} = c.in_{up} \land out_{up} = c.out_{up} \land inouts_{down} = c.inouts_{down}$

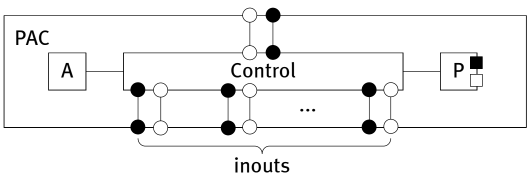


Figure L.2: Inside a PAC agent

PAC agents are loosely coupled with others. A dynamic hierarchical structure can be built up by connecting the exposed input and output ports through channels without any access to local state and operations.

BACKGROUND MATERIAL M

Specifications of the Actor

This background material presents the formal specification of the components of the Actor that is introduced in section 8.5 of chapter 8. An actor is basically a PAC agent that reacts on the user input events and scheduling commands, and takes actions to present media objects.

M.1 Refinement of *MediaObject*

Actors manage content elements, and the content elements are active media objects that prefetch and present the content data by themselves (see section F.8 on page 288). The content data is directly presented to the physical world $\mathbb{R}$ without separating the function from the interface. When PAC is applied for distributed applications, the presentation of the media object needs to be redirected to the presentation component of the actor, or, in a distributed setting, to be delegated by another actor. The delegation may happen when a real actor is connected to a virtual actor, the content of the media objects managed by the virtual actor should immediately redirected to the real actor. It may also happen if an actor (for example a satellite receiver) is used to decode the media, but needs another actor (for example a TV) to present the media.

Let's model the redirection or delegation of the presentation of a media object using the Streaming Channel pattern specified in previous background metrial. The class *MediaObject* is refined as follows:

MediaObject

$$\begin{array}{|l}
src : \mathbb{S} \\
p : \downarrow StreamSupplierImpl_{\copyright} \\
\hline
stype(src) = presentation.proxy.stream.type
\end{array}$$

$Retrieve \mathrel{\widehat=} [\, c? : \mathbb{C};\ d! : \mathbb{D}_{\circledcirc} \,]$

$Present = [\, d? : D_{\circledcirc} \,] \bullet \mathbf{if}\ d? \neq \mathsf{null}\ \mathbf{then}\ p.Push(d?)$

Every *MediaObject* has an output port *p* of the type *StreamSupplierImpl* that implements the *Push* interface of a *StreamSupplier*:

StreamSupplierImpl

StreamSupplier

$proxy : \downarrow ProxyStreamConsumer$

$Push \mathrel{\widehat=} proxy.Push$

Instead of presenting the data directly into the physical world $\mathbb{R}$, the operation *Present* pushes the data into a streaming channel through its output port *p*. The port *p* has a reference of a *ProxyStreamConsumer* object that is connected to the channel. Redirection or delegation of the presentation of the media object can then be done by connecting the presenting components, i.e. the *Presentation* component of the same agent or a delegating agent, to the same streaming channel using a *StreamConsumer* input port. Data received from the input port can then be presented to $\mathbb{R}$ through the *Present* operation of the *Presentation* component.

M.2 *ActorPresentation*

Based on the concepts of redirection and delegation of media object presentation, the *Presentation* component of an *Actor* should then include a set of input ports of the type *StreamConsumerImpl*:

ActorPresentation[*Input*, *Notification*, *Query*, *QueryResult*]

Presentation

$ins : \mathbb{P}\ StreamConsumerImpl_{\copyright}$

$\forall\, in : ins \bullet in.ap = self$

The type *StreamConsumerImpl* implements the *Push* interface of a *StreamConsumer* to passively receive the data from the streaming channel. Once the data is received, it is then presented by invoking the *Present* operation of the associated *ActorPresentation*:

StreamConsumerImpl

StreamConsumer

$ap : \downarrow ActorPresentation$

$Push \mathrel{\widehat=} [\, d? : \mathbb{D} \,] \bullet ap.Present(d?)$

Note that this is a minimalist description of the presentation behavior with regard to presentation redirection and delegation. In most of the cases of real implementation, the *Presentation* component might have different presentation behaviors for different types of data or the data received from different input ports. For example, MP3 streams would be presented to ℝ through the audio card and MPEG-4 streams would be through the graphics card using appropriate API invocation. Let's leave these details to the implementation.

M.3 *ActorAbstraction*

Although the concept of actions goes beyond just presenting media objects, here let's give an example specification of the *Abstraction* of component, where the core functions of an actor are to create and maintain media objects, and to schedule immediate or future actions on these media objects. These functions have already been covered in chapter 6 by various patterns. The *Abstraction* component is no more than a compositional realization of these patterns:

$ActorAbstraction[Input, Notification, Query, QueryResult, Command]$

$Abstraction$

$syncFactory : SyncFactoryImpl_{©}$
$mediaObjects : Dictionary$
$syncServiceFactory : SyncServiceFactoryImpl_{©}$
$proxies : Dictionary$

$HandleCommand \mathrel{\widehat=} [\, c? : \downarrow Command \,]$

An actor owns a *syncFactory* of the type *SyncFactoryImpl* that creates media objects for the given media types. Created objects are kept in a *Dictionary* of *mediaObjects*, which maps the name or the identification of created objects to the references of these objects. The actor also has a *syncServiceFactory* of the type *syncServiceFactoryImpl*, that wraps the created objects in *SyncService* objects and provides *SyncServiceProxy* objects so that synchronization services can be scheduled. Created *proxies* are kept in another *Dictionary*, with the same names or identifications mapped to the references of these created proxies.

An extra operation interface *HandleCommand* for the *ActorAbstraction* is also defined to receive scheduling commands from upper-level agents. Other operations of *ActorAbstration* that inherited from *Abstraction* need to be implemented to react on input events and scheduling commands, which is omitted from this specification. For a possible implementation, the example given in section G.8 on page 307 shows how these factories and their products can be used to create and maintain media objects, and to schedule and conduct the actions on these media objects.

M.4 *ActorControl*

PAC agents are mostly input event driven. From figure L.1 on page 361, one can see that the input events from the *Presentation* component update the internal state of the *Abstraction* component of the agent itself, and the *Abstraction* component of the upper-level agent. The user input may result in state changes thus trigger change notifications, these notifications are then sent through the *Control* horizontally to the *Presentation* component of the agent itself, or downwards to the *Presentation* component of lower-lever agents. This paradigm of input driven behavior is because the PAC pattern is originally designed as a structure for user interface toolkits and widgets.

An *Actor* is not only an interface component that is mainly controlled by the users, but also a media object presenter that must follow the scheduling and synchronization commands from upper-level agents. These commands can be a result of user interaction, but importantly, also a result of time that is triggered by an internal process of the scheduling agent.

Although these commands can be sent to lower level agents as change notifications, these notifications will first be sent to the presentation components according to the PAC pattern. The "notify-then-retrieve" scheme of updating the user interface does not fit the needs of scheduling and synchronization – for the sake of efficiency, these commands should reach the *Abstraction* component first instead of the *Presentation* component, which is a different scheme from the state change notifications.

To deal with these commands, the *Control* is refined as follows:

$ActorControl[Input, Notification, Query, QueryResult, Command]$

$Control$

$in_{up} : APushConsumerImpl_{\copyright}$

$ProcessCommand$

$c?, c_a!, c_{down}! : \downarrow Command_{\circledcirc}$

Decide whether to forward the command to *Abstraction*,
or to forward to lower-lever agents,
or both.

$HandleCommand \mathrel{\widehat=} ProcessCommand \parallel$
$\quad ([\, c_a?, c_{down}? : \downarrow Command_{\circledcirc} \,] \bullet$
$\quad\; (\mathbf{if}\; c_a? \neq \mathsf{null}\; \mathbf{then}\; c.HandleCommand(c?)) \parallel$
$\quad\; (\mathbf{if}\; c_{down}? \neq \mathsf{null}$

$$\textbf{then} \bigwedge in \mapsto out : inouts_{down} \bullet out.Push(c_{down}?)))$$

Two operations are added to the *Control* component to handle the scheduling commands. The *ActorControl* has to decide whether the received command needs to be sent to the associated *Abstraction*, or to be forwarded to lower-level agents. It is also possible that the command, for example a "stop all" command, needs to be sent to both the *Abstration* and the lower-level agents.

The input port of the *ActorControl* that connects the upper-level agent needs to be extended to direct the received commands to the operation *HandleCommand*:

APushConsumerImpl

$$[Input, Notification, Query, QueryResult, Command]$$

$$CPushConsumerImpl[Push_{pc} / Push]$$

$$\begin{aligned} Push \mathrel{\widehat=} Push_{pc} \ \| \\ & [\, d? : \mathbb{O} \,] \bullet \textbf{if}\, d? \in \downarrow Command\, \textbf{then}\, c.handleCommand(d?) \end{aligned}$$

BACKGROUND MATERIAL N

Firing rules of ASE

The firing rules of the ASE are summarized as follows:

1. Upon receiving a token, the token is locked and the place remains in the active state for the specified duration if the duration is determinable, or stays active until deactivated by the engine. During this period, the token is locked.

2. To enable a transition,

    - If a transition does not contain any priority place in its input places, the transition is immediately enabled when each if input places contains an unlocked token;
    - If a transition has at least one priority place in its input places, when the token in any one of the priority places becomes unlocked, the transition is immediately enabled.

3. When a transition is enabled,

    - If a transition does not contain any priority place in its input places, the transition removes a token from its input places.
    - If a transition has at least one priority place in its input places, a set of reverse tracking rules is followed to remove tokens from the input places.

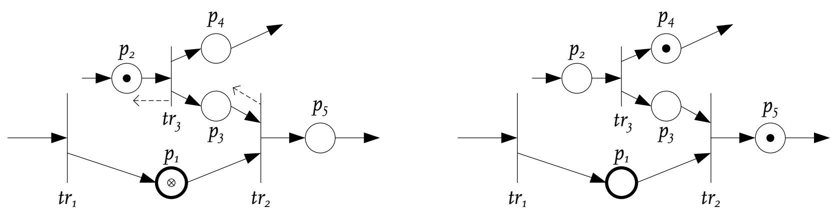
4. If an enabled transition is under the control of a runtime transition controller, the controller decides whether to fire the related transition instead or to fire both, and whether to update the structure between the controlled transition pairs.

5. The engine fires the enabled transition. Upon firing, the transition adds a token to each of its output places.

When a priority place enables a transition *tr*, there may exist some places that are before *tr* (perhaps even several steps before *tr*) and have tokens. These tokens become obsolete and need to be removed to restore the correct state of the model. One way to clear the obsolete tokens is to start from the initial place until *tr* is reached. This requires the paths between the initial place and the transition *tr* to be located first to avoid clearing the tokens in the places that are not in the paths. Yang and Huang (1996) propose a way to reversely track the paths to clear these obsolete tokens in their RTSM extension to OCPN, where reverse tracking in this case is thought to be more efficient since it avoids unnecessary traversal. The same approach is adopted in ASE. To remove tokens from the input places of *tr* in case of being enabled by a priority place *p*, the following reverse track rules should be used:

1. Reverse tracking stops at the starting transition of the priority place *p*, or at the initial place.
2. Along a path of the reverse tracking, if a place contains a token, either locked or unlocked, the token is removed and the reverse tracking is stopped in this path.
3. Along a path of the reverse tracking, if a transition is encountered, and this transition has output places that have not been reverse tracked, this transition is fired to activate these output places (not including the output places that have been tracked).

Figure N.1 shows an example of reverse tracking[1]. In figure N.1(a), a token in the priority place p_1 is unlocked and the transition tr_2 is enabled. Following the reverse tracking rules, the unlocked token in p_1 and the locked token in p_2 are removed. In the path of reverse tracking from p_3 to p_2, the transition tr_3 has an output place p_4 that has not been tracked, hence a token will be added to the place p_4. Moreover, the transition tr_2 is fired and a token is added to the place p_5. The resulting state of the model is shown in figure N.1(b).



(a) tr_2 is enabled by p_1 (b) tr_2 is fired

<--- reverse tracking ● locked token ⊗ unlocked token

Figure N.1: Reverse tracking upon a transition enabled by a priority place

[1]Please note that this reverse tracking has nothing to do with backtracking, cf. Prolog. It rather is a clean-up procedure to remove any pending tokens that are still underway towards a transition that got enabled because of a priority place. The strategy of not visiting places twice is well-known from other graph traversal problems such as garbage collection. The technique is to mark the places upon first visit.

BACKGROUND MATERIAL O

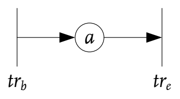
Converting IPML to ASE

This background material presents the translation schemes that convert the temporal constructs in IPML to ASE representations. The design principle adopted here is compositional translation. The conversion will first be focused on the basic temporal containers `seq` and `par`, and the basic temporal attributes `begin`, `end`, `dur` and `endsync`. The transition controllers are then used to convert other complex temporal attributes such as `max`, `restart`, `repeatCount`, `repeatDur` etc. Here also earns the fruits of the very early design decision that IPML is XML based: the development of the parser and the checker for IPML was a laborious but straightforward design subroutine. Therefore let's assume that the syntax of the IPML script has been checked and there is no semantic conflict in the script, in order to concentrate on the necessary conversions only.

During the description, it is necessary to refer to the concept of *syncbase*. In IPML timing, elements are timed relative to other elements. The syncbase for an element A is the other element B to which element A is relative. More precisely, it is the begin time or end time of the other element (Ayars et al., 2005). In ASE, the syncbase becomes a transition in the graph.

O.1 Action element

A simple action element in IPML that does not have any other temporal constraints can be directly converted to an ASE place that hosts the action. However when later converting the container elements, transitions as the beginning and ending points are needed to synchronize the containing elements. To keep the consistency of the elementary ASE constructs in the converting procedure, all converted ASE constructs are required to begin and end with transitions. The resulting ASE representation of a simple action element a is shown in figure O.1 on the next page.



```
<action id="a" />
```

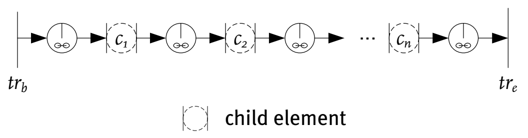
Figure O.1: Converting action to ASE

O.2 Timing containers

O.2.1 seq

A seq container defines a sequence of elements in which the elements are performed one after the other. The end time of a child element is taken by the next child element as the syncbase. The first child takes the begin time of its parent seq as the syncbase.

The child elements of the seq container can be action elements, or other synchronization containers (i.e., seq and par), so the conversion is a recursive procedure. The seq container can be converted into an ASE structure as shown in figure O.2, where the dashed circles with vertical bars at both sides are used to indicate the child elements that begin and end with transitions. Note that zero-duration connecting places are inserted between the children to keep the consistency, since an arc in ASE can only be used to link a transition and a place.



```
<seq>
    <... id="c1" ... />
    <... id="c2" ... />
    ...
    <... id="cn" ... />
</seq>
```

Figure O.2: Converting seq to ASE

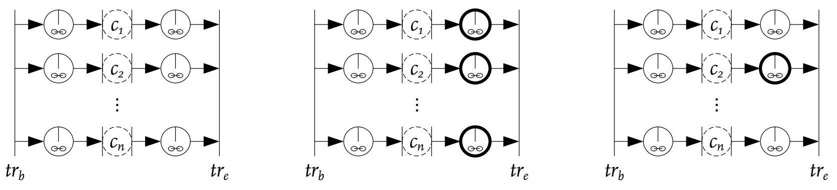
O.2.2 par

A par container, short for "parallel", defines a simple time grouping in which multiple elements can perform at the same time. All child elements take the begin time of their parent par as the syncbase.

Therefore, all child elements of a par container are put in parallel between two transitions as illustrated in figure O.3 on the facing page. There are three types of translations depending on the endsync attribute of the par container:

- If the endsync is "*last*", which is the default value, the par container is to end with the last end of all the child elements. The corresponding ASE is shown in figure O.3(a). The effect is that the transition tr_e is not enabled until all the child elements finish.

- If the endsync attribute is set to "*first*", the par is required to end with the earliest end of all the child elements. To convert to ASE, every child element is linked to the end transition with a priority connecting place as illustrated in figure O.3(b). The effect is that the child which ends first will immediately transit to the connected priority connecting place and in turn the transition tr_e will be enabled.
- If the value of endsync is an id attribute of one of the child elements, the par container is required to end with the specified element. In figure O.3(c), the connecting place between the specified element e_2 and the transition tr_e is set to a priority connecting place to implement the intended semantics.



(a) endsync="*last*" (b) endsync="*first*" (c) endsync="c_2"

child element

```
<par endsync="...">
    <... id="c1" ... />
    <... id="c2" ... />
    ...
    <... id="cn" ... />
</par>
```

Figure O.3: Converting par to ASE

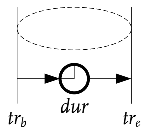
O.3 Timing attributes

O.3.1 dur

The attribute dur can be added to the container elements and the action elements to specify the explicit duration. Converting a dur to an ASE construct is straightforward: the value of the dur attribute, which is an interval time value, forms a priority timer place between the begin and end transitions of the element. Figure O.4 on the next page illustrates the effect of the attribute.

O.3.2 begin

The attribute begin specifies the time for the explicit begin of an action element or a container element. Two types of the begin values can be classified: offset values and event values. A time offset, for example "5s", is to postpone the performance of the element by 5 seconds from its syncbase. Hence, a priority timer place representing



dur="dur"

Figure O.4: Converting dur to ASE

the postponed duration can be added in front of the element as shown in figure O.5(a). For an event value, if the time can be determined during the converting process, it is converted to a time offset, otherwise it is considered to be nondeterministic.

Nondeterministic events include user interaction events such as "v_1.*activateEvent*", and synchronization events such as"v_1.*endEvent*" if the end time of the element v_1 can not be resolved when converting. Such nondeterministic events are converted to priority nondeterministic places as illustrated in figure O.5(b).

Event values can be combined with a time offset, for example "v_1.*activateEvent*+5*s*" represents the time of 5 seconds after v_1 is activated. The resulting ASE is a sequential composition of a nondeterministic place and a priority timer place (see figure O.5(c)).

The begin attribute may contain a list of values, separated with a ";", for example begin="5*s*; v_1.*activateEvent*; v_1.*beginEvent* + 5*s*". The converted places directly prior to the element are all priority places, which means ending of any of these places will start the element (see figure O.5(d)). As in this figure, the clouds will be used for including different sets of attribute values in other diagrams.

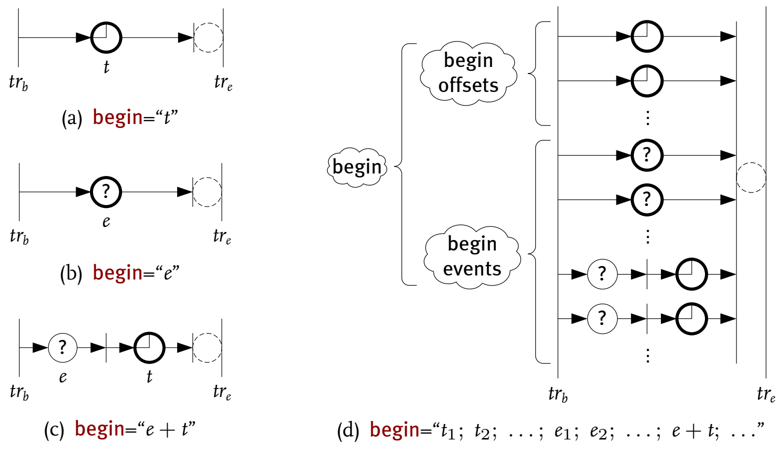


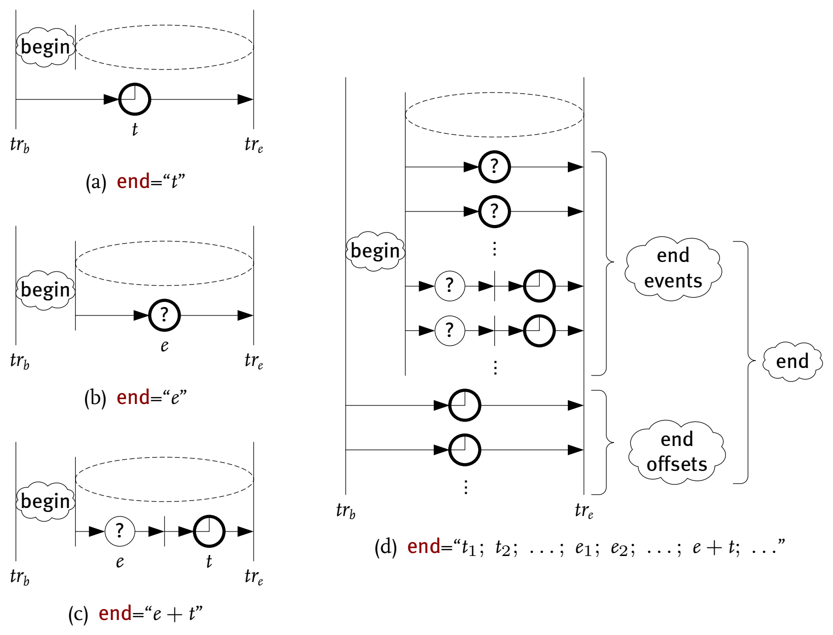
Figure O.5: Converting begin to ASE

O.3.3 end

The end attribute allows the author to constrain the active duration by specifying an end value using offset values or event values as in the begin attribute. In figure O.6(a), a priority timer place with the duration t is added to ensure the element will be ended after the time t, starting from the implicit syncbase of the element, no matter whether the element has been activated or not.

When the end attribute includes an event as the base, the event for ending can have an effect on the element only when the element is activated. Hence the events and the "event+offset" combinations are converted to structures that share the same input and output transitions with the element itself. Nondeterministic event values such as "v_1.*activateEvent*" are added as priority nondeterministic places as shown in figure O.6(b); event values with an offset such as "v_1.*beginEvent* $+ 5s$" are added as a sequential composition of a nondeterminacy place and a priority timer place as illustrated in figure O.6(c).

The end attribute may also contain a list of values, as in the begin. Figure O.6(d) illustrates an end value list converted to an ASE model.



(see figure O.5 for begin)

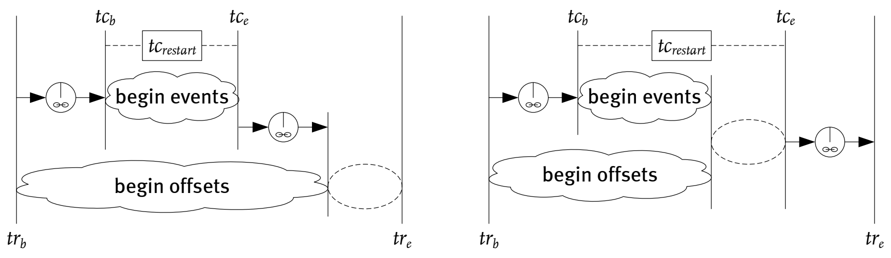
Figure O.6: Converting end to ASE

O.3.4 restart

IPML allows an element with begin specified to include event values to be restarted multiple times. This behavior is controlled by the restart attribute. A "restart" transition controller $tc_{restart}$ is used when the value of the restart attribute of an element is "*always*" or "*whenNotActive*" as shown in figure O.7(b) on the next page. First one should notice that in figure O.7(b), when a transition controller is to be added to a structure, a connecting place and a transition are inserted at both sides of the structure in a way that does not change the temporal semantics. The controller is then attached to the inserted transitions to avoid possible interference from other transition controllers – the new added controller will always be exclusively connected to the inserted transitions. Since the restart behavior is controlled only by the event values in the begin attribute, the begin attribute list is split into two parts: begin event values and begin offset values. They are treated differently when constructing the "restart" model. For different "restart" values, the resulting ASE model is different:

always The element can be restarted at any time. The transition controller is added around the list of the begin events as illustrated in figure O.7(a) such that when any of the begin events is triggered, the transition tc_e is enabled. The transition controller $tc_{restart}$ fires both tc_e and the linked tc_b, so that not only the element is started by tc_e immediately, but also at the same time, the places that hold the begin events are enabled by tc_b again. No matter whether the element is active or inactive, the begin events may "*always*" be triggered again and hence the element "restarts", until the entire structure is deactivated from the outside.

whenNotActive The element can only be restarted when it is no longer active. Attempts to restart the element during its active duration is ignored. The achieve this, the ASE model in figure O.7(b) adds the transition controller $tc_{restart}$ to cover both the begin events and the element in a way that the begin events and the element can not be enabled at the same time. When the active element becomes inactive, the transition tc_e is enabled. The transition controller $tc_{restart}$ fires both tc_e and tc_b. Firing of tc_e does not change the inactive state, but firing of tc_b enables the begin event places. Until the entire structure is deactivated, the events may again occur hence "restart" the element.



(a) restart="*always*" (b) end="*whenNotActive*"

(see figure O.5 for begin events and begin offsets)

Figure O.7: Converting restart to ASE

O.3.5 repeatCount and repeatDur

Repeating an element causes the simple duration to be activated several times in sequence. This will effectively copy or loop the contents of the action (or an entire structure in the case of a timing container). The author can specify either how many times to repeat, using repeatCount, or how long to repeat, using repeatDur (Ayars et al., 2005). Each repeat iteration is one instance of the active duration of the element. The repeatCount attribute specifies the number of iterations, which can have a positive numeric value or a value of "*indefinite*". When repeatCount is set to be "*indefinite*", the element is defined to repeat indefinitely, but subject to the constraints of the parent timing container. The repeatDur specifies the total duration for repeat.

As illustrated in figure O.8, if the element has a repeatCount, a "repeat" transition controller tc_{repeat} is added in parallel with the place of the element. If at the same time the element has a repeatDur attribute (say, with a value of *rdur*), a priority timer place with a duration of *rdur* is added in parallel as well. If the repeatDur is specified but the repeatCount attribute is not, a "repeat" transition controller is added anyway as if the repeatCount attribute is defined as "*indefinite*". When the transition tc_b is enabled, tc_{repeat} initiates a repeat counter with the value of repeatCount and records the current time instant of tc_b firing.

The priority connecting place *cp* is inserted after the place of the element to indirectly assign priority to the element place without knowing and changing its internal structure. Either the priority timer place of the repeatDur or the inserted priority connecting place is unlocked, the transition tc_e is enabled.

Once tc_e is enabled, the transition controller tc_{repeat} counts down the repeat counter by one step – but counting down does not have an effect on the value of "*indefinite*". The controller checks whether the counter is zero or the *repeatDur* is due. If yes, the controller removes the added *repeatDur* timer place and fires tc_e. Otherwise, it updates the duration of the *repeatDur* with remaining repeat duration, and fires the transition tc_b and records the time of tc_b firing for later calculating the remaining *repeatDur*.

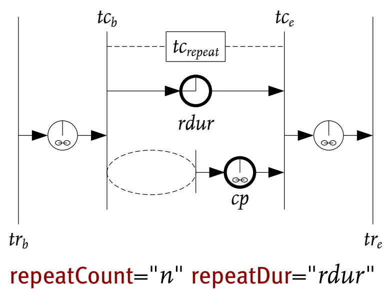


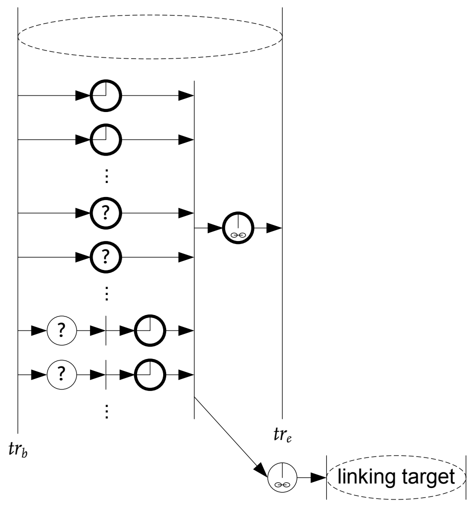
Figure O.8: Converting repeatCount and repeatDur to ASE

O.4 event based linking

Event based linking makes IPML very flexible in constructing non-linear narratives, especially for the situations where the user interaction decides the narrative directions during the performance. Multiple event linking elements can be added, so that a different set of events would link to different destinations.

A event linking element has an enable attribute that lists the interested event values. Note that the enable may also include a time offset value to trigger the link on the time offset to its syncbase. Multiple time offsets can be included but only the earliest one will take effect. Nevertheless, event linking elements are in most of the cases used for the links to be triggered by user interaction events.

Figure O.9 shows an event linking element in IPML converted to an ASE model. The enable values are converted into priority timer places and priority nondeterministic places in parallel to the element place. A transition is also inserted, on the one hand to trigger a priority connecting place to unlock the element place and fire the transition tr_e, on the other hand to transit to the linking target.



```
‹action id=" ... " href=" ... " actor=" ... "›
    ‹event enable="t1, t2, ..., e1, e2, ..., e + t, ..." href="linking target"›
‹/action›
```

Figure O.9: Converting event based linking to ASE

Although it is not shown in figure O.9, an event linking element may also have begin, end and dur attributes to enable the linking only during a specified interval. These attributes can be converted to ASE structures similar to the ones for action elements and containers.

O.5 ASE simplification

As mentioned earlier, some connecting places are inserted in the ASE model during the converting process to keep the consistency. Once the conversion is completed, the resulted ASE model can be simplified by removing some unnecessary connecting places. Three rules can be applied during this simplification process:

1. If the only output of a transition is an action element place and the only output is of the transition is a normal connecting place, the case can be naturally replaced with the action element place, because the connecting place does not contribute to anything.
2. If the only output of the transition in the previous rule is a priority connecting place, it means that the firing of the following transition of the action element

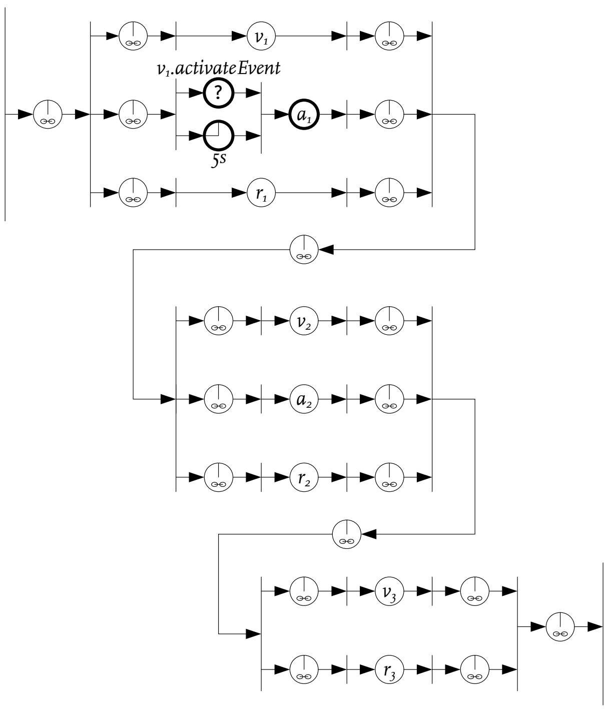


Figure O.10: An example conversion from IPML to ASE

place will immediately cause the firing of the following transition of the priory connecting place. The case can be replaced by changing the action element place to a priority one and remove the connecting place.

3. If there is only one connecting place between two transitions, the connecting place can be removed and the two transitions can be combined into one.

Note that these rules can apply only when the related transitions are not under control of any transition controller.

BACKGROUND MATERIAL **P**

Correctness of the mapping in section 10.1.4 of chapter 10

Here let's try to answer the following question: How to express the formal correctness of the implementation in section 10.1.4 of chapter 10?

Broy proposes three ideas of refinement: property refinement, glass box refinement, and interaction refinement. The glass box refinement is a classical concept of refinement typically used to decompose a system with a specified black box behavior into a distributed system architecture in the design phase, which seems appropriate in our case. The general form of a glass box refinement is

$$F_1 \oplus F_2 \oplus \cdots \oplus F_n \subseteq F$$

In Broy's theory (Broy, 1999), there is also another form for state machines, which is not needed here). The relation $\subseteq$ on component behaviors is defined by the rule that $\hat{F} \subseteq F$ stands for the proposition $\forall\, x : \vec{I} \bullet \hat{F}.x \subseteq F.x$, where

$$F : \vec{I} \to \mathcal{P}(\vec{O}), \hat{F} : \vec{I} \to \mathcal{P}(\vec{O}).$$

Also recall the definition of $\oplus$ for $F_1 \oplus F_2$

$$\begin{aligned}(F_1 \oplus F_2).x = \{&y \restriction O : y \restriction I = x \restriction I \\ &\wedge y \restriction O_1 = F_1(x \restriction I_1) \wedge y \restriction O_2 = F_2(x \restriction I_2)\}\end{aligned}$$

Let $RATE'$ be given by

$$\boxed{\begin{array}{l} rate(x.cable, 100, 100\text{K}) \\ rate(x.happy\_puppy, 10, 100) \\ rate(x.the\_sign, 2, 5\text{K}) \\ rate(x.button, 1, 1) \end{array}}$$

Note that this is essentially the same as $S_1, \ldots, S_4$ are saying about their z ports.

Consider an arbitrary x satisfying $RATE'$ where $x \in \overrightarrow{I'}$ where I' is $\{cable, \ldots\}$. x is a channel valuation $x : I' \to (M^*)^\infty$ such that

$$\forall\, i : I' \bullet x.i \in (type(i)^*)^\infty$$

where $M = \bigcup_{s \in S} s = PAL \cup MPG \cup DBH \cup IBH \cup \mathbb{R}$

Since I' has only four elements, let's write this out by assuming

$$\begin{array}{ll} x : cable \mapsto x_u & (x_u \in (PAL^*)^\infty) \\ x : happy\_puppy \mapsto x_d & (x_d \in (DBH^*)^\infty) \\ x : the\_sign \mapsto x_c & (x_c \in (MPG^*)^\infty) \\ x : button \mapsto x_b & (x_b \in (\mathbb{R}^*)^\infty) \end{array}$$

Now this x satisfies $RATE'$, that is, x_u has 100 frames per second, each frame being 100K bits, etc. (And similarly, x_d: 10, 100 respectively, x_c: 2, 5K respectively and x_b: 1, 1 respectively.)

Let $y \in SYSTEM'.x$ where $y \in \overrightarrow{O'}$ where O' is $\{screen, screen', moving, updown\}$ which can be written out by assuming

$$\begin{array}{ll} y : updown \mapsto y_i & (y_i \in (IBH^*)^\infty) \\ y : screen \mapsto y_s & (y_s \in (\mathbb{R}^*)^\infty) \\ y : screen' \mapsto y_{s'} & (y_{s'} \in (\mathbb{R}^*)^\infty) \\ y : moving \mapsto y_m & (y_m \in (\mathbb{R}^*)^\infty) \end{array}$$

The correctness requirement is

$$RATE' \wedge SYSTEM' \subseteq SYSTEM$$

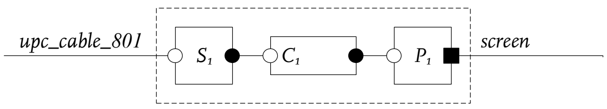
because $SYSTEM'$ only specifies maximal rates where as $SYSTEM$ has already fixed the rates.

The obvious proof strategy is to take an arbitrary x satisfying $RATE'$ and consider a $y \in SYSTEM'.x$, i.e.

$$y \in (STB' \oplus hi\_res\&PIP' \oplus \cdots) \setminus \{w_1, w_2, \ldots\}).x$$

and then check that $y \in SYSTEM.x$, which essentially boils down to checking *maxdelay* constraints and checking the essential transformations of the form pr'' for each of the specification paths, e.g. when going from *cable* to *screen*, or when going from *the_sign* to *screen'*.

As an example let's check that y_s satisfies all constraints of $SYSTEM.x$. Since $SYSTEM$ falls apart in 4 unconnected parts, it is enough to check that $y_s = y.screen$ satisfies the the constraints of $(S_1 \circ C_1 \circ P_1)\setminus\{z, w\}$:



$$\begin{array}{l} \exists\, z : (PAL^*)^\infty \bullet \exists\, w : (PAL^*)^\infty \bullet \\ \quad z = x.cable \wedge rate(z, 100, 100\text{K}) \\ \quad \wedge\ maxdelay(z, w, 2) \wedge y.screen = pr''(w) \end{array}$$

which is equivalent to:

$$rate(x.cable, 100, 100\text{K}) \wedge maxdelay(pr''(x.cable), y.screen, 2)$$

(since pr'' works frame-wise.)

What is known about y_s comes from $RATE' \wedge SYSTEM'$, where $RATE'$ means $rate(cable, 100, 100\text{K})$. For $SYSTEM'$, its meaning is more complicated. First, note that

$$SYSTEM' : \overset{+}{I} \to \mathcal{P}(\overset{+}{O})$$

also, $SYSTEM' = (STB' \oplus hi\_res\&PIP' \oplus \cdots)$, where

$$\begin{array}{l} STB' = STB[cable/channel, w1/v, \ldots] \\ \quad \ll control : channel2v\_x2exec \setminus \{control, x, y\} \\ hi\_res\&PIP' = hi\_res\&PIP[w1/x] \\ \quad \ll control : x2screen \setminus \{control, u, PIP\} \end{array}$$

Note that for y_s it suffices to focus on STB' and $hi\_res\&PIP$ and forget about all other channels than w_1 where $\oplus$ can be replaced by $\circ$, i.e.

$$SYSTEM' = hi\_res\&PIP' \circ STB'$$

Let's summarize the assertions from $SYSTEM'$ by performing the renamings and keeping only the relevant clauses in view of the chosen control command. STB' says:

$$\begin{array}{l} maxrate(x.cable, 100\text{M}) \\ \quad \wedge\ delay(x.cable, w_1 \copyright PAL, \underbrace{\Delta_{STB}}_{1}) \ ; \end{array}$$

$hi\_res\&PIP'$ says

$$\begin{array}{l} maxrate(w_1, 100\text{M}) \\ \quad \wedge\ delay(pr''(w_1 \copyright PAL), y.screen, \underbrace{\Delta_{hi\_res\&PIP}}_{1}) \ ; \end{array}$$

$rate(x.cable, 100, 100\text{K})$ decides that for all t

$$\#(x.cable.t) = 100$$

and for each i, the i-th frame has

$$\#bits(x.cable.t.i) = 100\text{K},$$

therefore

$$\sum_{i=1}^{100} \#bits(x.cable.t.i) = 10\text{M}$$

so the *maxrate* requirement of STB' saying

$$\sum_{i=1}^{\#(cable.t)} \#bits(cable.t.i) < \text{100M}$$

is not contradicted.

Next let's check the delays. Let's combine the two one-step delay assertions, first by noting that pr'' works frame-wise:

$$\begin{cases} delay(pr''(x.cable), pr''(w_1 \copyright PAL), 1) \\ delay(pr''(w_1 \copyright PAL), screen, 1) \end{cases}$$

and secondly by adding the delays:

$$delay(pr''(x.cable, y.screen, 2)$$

which satisfies the clause from $SYSTEM$ that

$$maxdelay(pr''(x.cable), y.screen, 2)$$

as required. The other $SYSTEM$ constraints can be checked in a similar manner.

Glossary

A

Abstract Windowing Toolkit (AWT) Java's platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is part of the JFC - the standard API for providing a graphical user interface (GUI) for a Java program.

Action Synchronization Engine (ASE) A runtime synchronization Engine that takes the timing and synchronization relations defined in an IPML script as input, and creates an object-oriented representation based on an extended version of OCPN.

Ambient Intelligence (AmI) The concept of ambient intelligence or AmI is a vision where humans are surrounded by computing and networking technology unobtrusively embedded in their surroundings. It refers to digital environments that are sensitive, adaptive, and responsive to the presence of people. In such a environment, ambient intelligence will improve the quality of life of people by creating the desired atmosphere and functionality via intelligent, personalized inter-connected systems and services.

Application Programming Interface (API) a set of routines, protocols, and tools prescribed by a computer operating system or by an application program by which a programmer writing an application program can make requests of the operating system or another application.

Architecture Based Design (ABD) A method for designing the high-level software architecture for complex systems with detailed requirements unknown in advance. The ABD method fulfills functional, quality, and business requirements at a level of abstraction that allows for the necessary variation when producing specific products. Its application relies on an understanding of the architectural mechanisms used to achieve this fulfillment.

Artificial Intelligence (AI) Ability of a machine to perform tasks thought to require human intelligence. Typical applications include game playing, language translation, expert systems, and robotics.

Audio Video Interleave (AVI) A multimedia container format introduced by Microsoft,containing both audio and video data in a standard container that allows simultaneous playback.

B

Binary Format for Scenes (BIFS) A binary format for two- or three-dimensional audiovisual content. It is based on VRML and is part of the MPEG-4 standard.

C

Central Processing Unit (CPU) the component in a digital computer that interprets and executes instructions and data contained in software.

Common Intermediate Language (CIL) The lowest-level human-readable programming language in the Microsoft .NET (.NET) Framework. Languages which target the .NET framework compile to CIL, which is assembled into bytecode and executed by a virtual machine.

Common Object Request Broker Architecture (CORBA) a standard for software componentry created and controlled by the Object Management Group (OMG). It defines APIs, communication protocol, and object/service information models to enable heterogeneous applications written in various languages running on various platforms to interoperate.

Communicating Sequential Processes (CSP) A formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi.

Component Object Model (COM) A Microsoft platform as a software component technology. It is used to enable interprocess communication and dynamic object creation in any programming language that supports the technology.

Computer Supported Cooperative Work (CSCW) CSCW addresses how collaborative activities and their coordination can be supported by means of computer systems. CSCW is also often referred as computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment.

D

Digital Theater Systems (DTS) A multi-channel surround sound format used for both commercial and consumer grade applications (with slight technical differences between home and commercial variants). It is primarily used for in-movie sound both on film and on DVD. The company which created it, Digital Theater Systems, is also often referred to simply as DTS.

Distributed Component Object Model (DCOM) A Microsoft proprietary technology for software components distributed across several networked computers to communicate with each other. It extends Microsoft's Component Object Model (COM), and provides the communication substrate under Microsoft's COM+ application server infrastructure.

Distributed Computing Environment (DCE) A software system developed in the early 1990s by a, consortium that included HP, IBM, DEC, and others. The DCE supplies a framework and toolkit for developing client/server applications. The framework includes a remote procedure call (RPC) mechanism known as DCE/RPC, a naming (directory) service, an authentication service, and a distributed file system (DFS) known as DCE/DFS.

Distributed Computing Environment/Remote Procedure Calls (DCE/RPC) see Distributed Computing Environment (DCE).

Document Object Model (DOM) A platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented document.

Document Type Definition (DTD) A declaration in an SGML or XML document that specifies constraints on the structure of the document, describes each allowable element within the document, the possible attributes and (optionally) the allowed attribute values for each element.

F

First In, First Out (FIFO) A queue or a system in which the first item stored is the first item retrieved.

G

General Inter-ORB Protocol (GIOP) In distributed computing, GIOP is the abstract protocol by which the Object Request Broker (ORB) components communicate. Standards associated with the protocol are maintained by the Object Management Group (OMG).

GET An action HTTP defines to be performed on the identified resource. It requests a representation of the specified resource. See also POST.

Graphical User Interface (GUI) A method of interacting with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text.

H

Home Audio Video Interoperability (HAVi) A Consumer Electronics (CE) industry standard that ensures interoperability between digital A/V devices from different vendors and brands that are connected using IEEE 1394 (FireWire).

Home Network Broker (HNB) A software agent in VESA that helps one device to discover other devices, organizes device names into a hierarchical namespace, organizes interfaces of devices into one searchable interface repository or a distributed interface repository, and offers device interfaces from the interface repository to an interface requester.

hypermedia Hypermedia is a term used as a logical extension of the term hypertext, in which audio, video, plain text, and non-linear hyperlinks intertwine to create a generally non-linear medium of information.

HyperText Markup Language (HTML) A markup language designed for the creation of web pages with hypertext and other information to be displayed in a web browser. HTML is used to structure information – denoting certain text as headings, paragraphs, lists and so on. It can be used to describe, to some degree, the appearance and semantics of a document.

HyperText Transfer Protocol (HTTP) A request/response protocol between clients and servers on the Web. An HTTP client, such as a web browser, typically initiates a request by establishing a TCP connection to a particular port on a remote host. An HTTP server listening on that port waits for the client to send a request string. Upon receiving the request, the server sends back a response string and the body of the message which can be the requested file, an error message, or some other information.

I

ICE-CREAM stands for "Interactive Consumption of Entertainment in Consumer Responsive, Engaging & Active Media", a project funded by Information Society Technologies programme of the European Community to investigate the potential of new technologies, e.g., MPEG-4 and Internet, for designing new concepts for interactive and enhanced broadcast, to create programmes and environments in which users can interact with the content of the programmes and compose personal programmes to create, with simple tools, personal flavor and emotion, i.e. to create their personal immersive experience (ICE-CREAM, 2003). The project started in January 2002 and finished in December 2003.

IEEE 1394 (FireWire) A personal computer and digital video serial bus interface standard offering high-speed communications and isochronous real-time data services. FireWire can be considered a successor technology to the obsolescent SCSI Parallel Interface. Up to 63 devices can be daisy-chained to one FireWire port.

Interactive Play Markup Language (IPML) A markup language designed for rendering interactive plays with multiple connected presentation devices in a distributed environment, based on SMIL.

Interactive Story Markup Language (StoryML) An XML based scripting language for specifying interactive stories based on the concept of parallel storylines, where interaction causes switching among these sotrylines. It depicts the requiremnents of presenting environment by listing the types of actors that are needed, instead of directly naming these actors. A StoryML script must be presented in an evironment driven by a StoryML playback sytem, and the word StoryML also often refers to such a system.

Internet Inter-ORB Protocol (IIOP) Implementation of GIOP for TCP/IP. It is a concrete realization of the abstract GIOP definitions.

Internet Protocol (IP) A data-oriented protocol used by source and destination hosts for communicating data across a packet-switched network. Devices identify themselves using a unique IP address.

Inversion of Control (IoC) Also called Dependency Injection. A popular Object-oriented programming principle which inverts the way an object gets its dependencies. An object has dependencies: other objects it needs in order to function. Either it creates the object it needs internally or it is given to it by another object. IOC inverts this pattern. It checks what an object needs in order to function, and “injects" those dependencies into the object.

ITC-SOPI The Television Commission Sense Of Presence Inventory (ITC-SOPI), from the UK Independent Television Commission.

J

Java Foundation Classes (JFC) A graphical framework for building portable Java-based graphical user interfaces (GUIs). JFC consists of the Abstract Windowing Toolkit (AWT), Swing and Java2D. Together, they provide a consistent user interface for Java programs, regardless whether the underlying user interface system is Windows, Mac OS X or Linux.

Java Media Framework (JMF) An application programming interface (API) from Sun Microsystems, Inc., for incorporating time-based media into Java applications and applets.

Java Virtual Machine (JVM) A virtual machine serves as interpreter between Java bytecode and a specific operating system that allows Java applications to run on any platform without changing the code.

JINI an open software architecture that enables Java Dynamic Networking for building distributed systems that are highly adaptive to change. Jini technology can be used to deliver adaptive technology systems that are scalable, evolvable, and flexible, as typically required in dynamic runtime environments.

L

Language Of Temporal Ordering Specification (LOTOS) A formal specification language based on temporal ordering used for protocol specification in ISO OSI standards. It was published as ISO 8807 in 1990 and describes the order in which events occur.

Linux Infrared Remote Control (LIRC) LIRC is a package that allows to decode and send infra-red signals of many (but not all) commonly used remote controls, developed by an open source project for adding IR remote control(TV-remote) support to Linux computers.

Logical Data Unit (LDU) In multimedia systems, time-dependent objects usually consist of a sequence of information units, often referred as *Logical Data Units (LDU's)*, such as audio samples or video frames.

M

Microsoft COM+ (COM+) An extension to Component Object Model (COM) introduced by Microsoft in Windows 2000. The advantage of COM+ was that it could be run in "component farms", managed with the built-in Microsoft Transaction Server.

Microsoft Foundation Classes (MFC) A Microsoft library that wraps portions of the Windows API in C++ classes, forming an application framework. Classes are defined for many of the handle-managed Windows objects and also for predefined windows and common controls.

Microsoft Media Services (MMS) MMS is Microsoft's proprietary network streaming protocol. MMS protocol can be used on top of TCP or UDP transport protocols over any network medium.

Microsoft .NET (.NET) A framework created by Microsoft as a software development platform similar to Java technology. At the core is the use of a virtual machine that runs a bytecode called Common Intermediate

Language (CIL). Programs are compiled to produce CIL and then CIL is distributed to users to run on a virtual machine.

Model-View-Controller (MVC) A model, therefore is an object representing data or even activity, e.g. a database table or even some plant-floor production-machine process. A view is some form of visualization of the state of the model. A controller offers facilities to change the state of the model.

MP3 A popular digital audio encoding and lossy compression format invented in 1987 by the Fraunhofer Institute for Integrated Circuits" in Germany. The name is derived from "MPEG-1 Audio Layer 3", more formally known as "MPEG-1 Part 3 Layer 3" or "ISO/IEC 11172-3 Layer 3".

MPEG-2 A group of audio and video coding standards agreed upon by MPEG (Moving Pictures Experts Group), and published as the ISO/IEC 13818 international standard. MPEG-2 is typically used to encode audio and video for broadcast signals, including direct broadcast satellite and Cable TV. MPEG-2, with some modifications, is also the coding format used by standard commercial DVD movies.

MPEG-4 An ISO/IEC standard developed by MPEG (Moving Picture Experts Group), provides the standardized technological elements enabling the integration of the production, distribution and content access paradigms of digital television, interactive graphics applications and interactive multimedia.

multimedia A term used to describe multiple means of media which are used to convey information (text, audio, graphics, animation, video, and interactivity). As the information is presented in various formats, multimedia enhances user experience and helps grasping information better and faster.

Multimedia Home Platform (DVB-MHP) An open middleware system standard designed by the DVB project for interactive digital television. The MHP enables the reception and execution of interactive, Java-based applications on a TV-set. Applications can be delivered over the broadcast channel, together with audio and video streams. These applications can be for example information services, games, interactive voting, e-mail, sms or shopping. For all interactive applications an additional return channel is needed.

Multimedia Messaging System (MMS) A system of transmitting not only text messages, but also various kinds of multimedia contents (e.g. images, audio and/or video clips) over wireless networks using the Wireless Application Protocol (WAP) protocol.

Multipurpose Internet Mail Extensions (MIME) MIME is an Internet Standard for the format of e-mail. MIME defines a collection of e-mail headers for specifying additional attributes of a message including the content type which indicates the type and subtype of the message content, for example, Content-type: text/plain. The combination of type and subtype is generally called a MIME type, although in modern applications, Internet media type is the favored term, indicating its applicability outside of MIME messages.

N

Network File System (NFS) A protocol originally developed by Sun Microsystems as a distributed file system which allows a computer to access files over a network as easily as if they were on its local disks.

Network Information System (NIS) A network naming and administration system for smaller networks, developed by Sun Microsystems. It is a RPC-based client/server system that allows a group of machines within an NIS domain to share a common set of configuration files. This permits a system administrator to set up NIS client systems with only minimal configuration data and add, remove or modify configuration data from a single location.

Network Time Protocol (NTP) A network protocol that schedules the computer's internal clock with the atomic clocks or radio clocks on the Internet.

NexTV stands for "New media consumption in EXtended interactive TeleVision environment", a project funded by Information Society Technologies programme of the European Community to investigate how the new interactive technologies such as MPEG-4, XML and DVB-MHP can influence the traditional television broadcasting (NexTV, 2001). NexTV commenced in January 2000 and finished in December 2001.

O

Object Composition Petri Net (OCPN) A formal specification and modeling technique for multimedia composition with respect to inter-media timing. The model is based on the logic of temporal intervals, and Timed Petri Nets.

Object Request Broker (ORB) In distributed computing, an object request broker (ORB) is a piece of middleware software that allows programmers to make program calls from one computer to another, via a network. ORBs handle the transformation of in-process data structures to the byte sequence which is transmitted over the network and also the reverse transformation. In the object oriented languages, the ORB is an object, having methods to connect the objects being served. After such object is connected to the ORB, the methods of that object become accessible for remote invocations.

Open Network Computing (ONC) An industry standard developed by the Sun Corporation, which defines the use of Remote Procedure Calls (RPC) among connected systems.

Open Services Gateway Initiative (OSGi) A non-profit corporation that created an open specification for delivery of a gateway for services like energy measurement and control, safety and security, health care monitoring, device control and maintenance and e-commerce.

P

POST An action HTTP defines to be performed on the identified resource. It submits user data (e.g. from a HTML form) to the identified resource. The data is included in the body of the request. See also GET.

Presentation-Abstraction-Control (PAC) The Presentation-Abstraction-Control architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the applications functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Presentation Environment for Multimedia Objects (PREMO) An ISO standard to provide a standardised development environment for multimedia applications (ISO/IEC JTC 1/SC 24). It concentrates mainly on presentation techniques. PREMO is designed to work with existing and emerging standards.

Q

Quality of Service (QoS) On the Internet and in other networks, QoS, is the idea that transmission rates, error rates, and other characteristics can be measured, improved, and, to some extent, guaranteed in advance. QoS is of particular concern for the continuous transmission of high-bandwidth video and multimedia information. Transmitting this kind of content dependably is difficult in public networks using ordinary "best effort" protocols.

R

Real Time Streaming Protocol (RTSP) Developed by the IETF and published in 1998 as RFC 2326, RTSP is a protocol for use in streaming media systems which allows a client to remotely control a streaming media server, issuing VCR-like commands such as "play" and "pause", and allowing time-based access to files on a server.

Real-time Transport Control Protocol (RTCP) Defined in RFC 3550, RTCP provides out-of-band control information for an RTP flow. It partners RTP in the delivery and packaging of multimedia data, but does not transport any data itself. It is used periodically to transmit control packets to participants in a streaming multimedia session. The primary function of RTCP is to provide feedback on the quality of service being provided by RTP.

Real-time Transport Protocol (RTP) RTP defines a standardized packet format for delivering audio and video over the Internet. It was developed by the Audio-Video Transport Working Group of the IETF and first published in 1996 as RFC 1889.

Remote Method Invocation (RMI) A Java application programming interface for performing remote procedural calls.

Remote Procedure Call (RPC) A protocol that allows a computer program running on one host to, cause code to be executed on another host without the programmer needing to explicitly code for this. When the code in question is written using object-oriented principles, RPC is sometimes referred to as remote invocation or remote method invocation.

Resource Description Framework (RDF) a family of specifications for a metadata model that is often implemented as an application of XML. The RDF family of specifications is maintained by the World Wide Web Consortium (W3C). The RDF metadata model is based upon the idea of making statements about resources in the form of a subject-predicate-object expression, called a triple in RDF terminology. The subject is the resource, the "thing" being described. The predicate is a trait or aspect about that resource, and often expresses a relationship between the subject and the object. The object is the object of the relationship or value of that trait.

S

Simple Object Access Protocol (SOAP) SOAP is a protocol, defining XML-based information, which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment over a computer network, normally using HTTP.

Simple Service Discovery Protocol (SSDP) a discovery service allows a Windows computer to discover universal plug and play devices on a network.

Standard Generalized Markup Language (SGML) A metalanguage in which one can define markup languages for documents, provides a variety of markup syntaxes that can be used for many applications. SGML is very flexible and powerful, but its complexity has prevented its widespread application for small-scale general-purpose use. HTML and XML are both derived from SGML. While HTML is an application of SGML, XML is a profile – a specific subset of SGML, designed to be simpler to parse and process than full SGML.

Synchronized Multimedia Integration Language (SMIL) A recommendation from W3C (The World Wide Web Consortium). It enables simple authoring of interactive audiovisual presentations. SMIL is typically used for "rich media"/multimedia presentations which integrate streaming audio and video with images, text or any other media type.

T

Tangible User Interface (TUI) A user interface in which a person interacts with digital information through the physical environment. Tangible interfaces give physical form to digital information, employing physical artifacts both as representations and controls for computational media.

Text-to-Speech (TTS) Artificial production of human speech based on text. A system used for producing human speeck is often called a speech synthesizer, and can be implemented in software or hardware. Speech synthesis systems are often called text-to-speech (TTS) systems in reference to their ability to convert text into speech.

Timed Communication Object-Z (TCOZ) An integration of Object-Z and Timed CSP. The approach taken in TCOZ is to identify operation schemas (both syntactically and semantically) with (terminating) CSP processes that perform only state update events; to identify active classes with non-terminating CSP processes; and to allow arbitrary (channel-based) communications interfaces between objects.

Transmission Control Protocol (TCP) One of the core protocols of the Internet protocols. TCP is the intermediate layer between the Internet Protocol (IP) below it, and an application above it. Using TCP, applications on networked hosts can create connections to one another, over which they can exchange data. The protocol guarantees reliable and in-order delivery of sender to receiver data. TCP does the task of the transport layer in the OSI model of computer networks.

U

Unified Modeling Language (UML) An open method used to specify, visualize, construct and document, the artifacts of an object-oriented software-intensive system under development. The UML represents a compilation of best engineering practices which have proven to be successful in modeling large, complex systems, especially at the architectural level.

Uniform Resource Identifier (URI) An Internet protocol element consisting of a short string of characters that conform to a certain syntax. The string comprises a name or address that can be used to refer to a resource. It is a fundamental component of the World Wide Web.

Uniform Resource Locator (URL) A standardized address for some resource (such as a document or image) on the Internet (or elsewhere). First created by Tim Berners-Lee for use on the World Wide Web, the currently used forms are detailed by Internet standard RFC 3986.

Universal Plug and Play (UPnP) A set of computer network protocols to allow devices to connect seamlessly and to simplify the implementation of networks in the home and corporate environments. UPnP achieves this by defining and publishing UPnP device control protocols built upon open, Internet-based communication standards.

User Datagram Protocol (UDP) A protocol with no connection required between sender and receiver that allows sending of data packets (known as datagrams) on the Internet (thought unreliable because it cannot ensure the packets will arrive undamaged or in the correct order).

User Interface Markup Language (UIML) An XML language for defining user interfaces on computers. The standard can be found at http://uiml.org.

V

Video Electronic Standards Association (VESA) A group of manufacturers which develops standards for graphics cards; manager of the standard for display cards which enables work in Super VGA mode.

W

Web Ontology Language (OWL) A markup language for publishing and sharing data using ontologies on the Internet. OWL is a vocabulary extension of RDF and is derived from the DAML+OIL Web Ontology Language. Together with RDF and other components, these tools make up the semantic web project.

X

eXtensible HyperText Markup Language (XHTML) A markup language that has the same expressive possibilities as HTML, but a stricter syntax. Whereas HTML is an application of SGML, a very flexible markup language, XHTML is an application of XML, a more restrictive subset of SGML,.

Extensible Markup Language (XML) A simple, flexible text format derived from SGML, (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.

Index

$\mathbb{C}$, synchronization coordinates, 278
$\mathbb{D}$, data, 289
$\mathbb{S}$, source of data, 289
$\mathbb{T}$, basic time type, 269
$\__{\circledcirc}$, Free type with the constant null, 274
$\upharpoonright$, sequence filtering, 330
$\oplus$, relational overriding, 267
$\triangleright$, is a, 303
$\mathbb{R}$, real number, 133, 136, 290
$\mathbb{R}$, "things in the real world", 145, 289
τ, absolute current time, 269
Object-Z
- !, output decoration, 62
- $'$, change decoration, 62
- ?, input decoration, 62
- $\bullet$, scope enrichment, 269
- $\__{\copyright}$ object containment, 273
- Δ, Δ-list, 62
- $\__{\text{id}}$ global object indenty, 325
- Init, initial state, 62
- Process, active process, 270
- $:=$, operation to assign a value to a state variable, 63
- $\bigwedge$, operation conjunction, 62
- $[]$, angelic choice, 270
- $\|$, parallel operation composition, 287
- $\|_!$, parallel operation composition, output exposed, 304
- $⨟$, sequential operator, 63
- $\setminus$, hiding, 269
- $\mathcal{O}$, object wrapper for primary values, 266
- $\mathbb{O}$, object universe, 266
- $\mathcal{V}$, original value of the wrapped object, 266
- $\downarrow$, polymorphic notation, 62
- *self*, self identity, 62
- $\rightsquigarrow$, sequential composition that outputs and inputs values, 63
- $\upharpoonright$, visibility list, 269
- delete, object destruction, 273
- **if** . . . **then** . . . **else** . . ., conditional composition, 283
- new, object creation, 273

in, in the interests described by, 325
matches, string matching, 325
x masculinity/femininity, 190

ABD, 6
Abstract Factory, 74
Abstraction, 357
AbstractSyncFactory, 304
AbstractSyncServiceFactory, 306
Action, 267
action, 39, 42, 48, 373
Action Service Factory, 73

- *AbstractSyncFactory*, 304
- *AbstractSyncServiceFactory*, 306
- *ReadyAction*, 302
- *StartAction*, 302
- *SyncFactoryImpl*, 305
- *SyncService*, 301
- *SyncServiceFactoryImpl*, 307
- *SyncServiceProxy*, 302

Action Synchronization Engine, 136
action time, 120
ActionService, 265
ActionServiceProxy, 272
Active Object, 68
ActiveController, 355
ActiveEventDispatcher, 299
ActivePresentation, 360
ActiveSynchronizable, 288
ActiveSyncMediaObject, 290
Actor
- *ActorPresentation*, 366
- *MediaObject*, 365
- *StreamConsumerImpl*, 366, 367
- *StreamSupplierImpl*, 366

actor, 30, 39, 42, 46, 113
ActorPresentation, 366
adapted time, 120
agreeableness, 190
ambient intelligence, 1
anonymous operation, 62
architectural construct, 59
architecture, 6
Architecture-based design, 6
ASE, 136
Attention, 173

begin, 375
BIFS, 26

Challenge, 173
Channel, 80
- *Channel*, 311
- *ConsumerAdmin*, 313
- *ProxyPullConsumer*, 320
- *ProxyPullSupplier*, 317
- *ProxyPushConsumer*, 319
- *ProxyPushSupplier*, 315
- *PullConsumer*, 320
- *PullSupplier*, 316
- *PushConsumer*, 318
- *PushSupplier*, 315
- *SupplierAdmin*, 312

Channel, 311
Command, 267
ConsumerAdmin, 313
content element, 42, 43
- time-dependent element, 118
- time-independent element, 118

content element time, 119
context dependent interaction, 18
Control, 173
Control, 361
Controller, 354
cooperative interaction, 23
coordinate, 277
CPushConsumerImpl, 363
culture dimensions, 190
- individualism/collectivism, 190
- long/short-term orientation, 190
- masculinity/femininity, 190
- power distance, 190
- uncertainty avoidance, 190

Curiosity, 173

DataCache, 340
DataInterest, 325
DataSink, 340
DataSource, 341
DataSourceReceiver, 341
decision points, 16
Decorator, 59, 292
DeepSea, 20
design pattern, 53
dialog, 29
Dictionary, 265
dimensions of culture, 190
direction, 279
director, 39, 42, 113
Dispatcher, 331
distributed interfaces, 2, 17
distributed multimedia, 2
Document − View, 105
dur, 375

Easy Access, 1
Ecological Validity, 174
embodied characters, 192
embodiment, 192
end, 376
Engagement, 174
Enjoyability, 173
entertainment, 173
entertainment experience, 173
environment, 30
Event, 48
Event, 279
event based linking, 379
EventDispatcher, 298
EventHandler, 299
ExecutionQueue, 268
experience, 173

Factory Method, 74, 274
feed-forward, 29
feedback, 29
Filter, 324
fun, 173
fun experience, 173
    Attention, 173
    Challenge, 173
    Control, 173
    Curiosity, 173
    Enjoyability, 173
    Presence, 174
    Pride, 174
Future, 275
Future TV, 27

Get, 267

harmony, 190
harmony in six aspects, 190
HomeLab, 1
HTML, 29

ICE-CREAM, 19
Immersive broadcast, 28
individualism/collectivism, 190
instant, 122
inter-media synchronizatio, 121
interactive play, 42
interval, 125
intra-media synchronization, 121
IPML, 45
    action, 48, 373
    actor, 46
    begin, 375
    dur, 375
    end, 376
    Event, 48
    event based linking, 379
    par, 374
    repeatCount, 378
    repeatDur, 378
    restart, 377
    seq, 374
    syncbase, 373
ITC-SOPI, 175

line, 39
long/short-term orientation, 190

$M - VC$, 105
mapping, 40
masculinity/femininity, 190
$MC - V$, 105
MDPushConsumerImpl, 353
media form, 174
media objects, 30
Media types, 303
MediaObject, 289, 365
Model, 349
ModelDependent, 352
Monitor, 68
MPEG-4, 26
MPushConsumerImpl, 351
multimedia, 2
MultiWindowLayout, 27
MVC, 32, 102, 106
    ActiveController, 355
    ActivePresentation, 360
    Controller, 354
    $Document - View$, 105
    $M - VC$, 105
    $MC - V$, 105
    MDPushConsumerImpl, 353
    Model, 349

    ModelDependent, 352
    MPushConsumerImpl, 351
    PushSupplierImpl, 352
    View, 354

natural temporal relation, 118
Negative Effects, 174
NexTV, 11

Observable, 62
Observer, 55, 62
    Observable, 62
    Observer, 62
Observer, 62
OCPN, 126, 132

PAC, 32
    Abstraction, 357
    Control, 361
    CPushConsumerImpl, 363
    PAC, 364
    Presentation, 359
PAC, 364
par, 374
pattern, 53
pattern catalog, 61
pattern language, 55
Patterns
    Abstract Factory, 74
    Action Service Factory, 73
    Active Object, 68
    Channel, 80
    Command, 267
    Decorator, 59, 292
    Factory Method, 74, 274
    Future, 275
    Monitor, 68
    MVC, 102, 106
    Observer, 55, 62
    Plugin, 92
    Proactor, 72, 275
    Proxy, 272
    Reactor, 71, 298
    Real-time Channel, 88
    Streaming Channel, 93
    Synchronizable Object, 69
    Timed Action, 65

performance, 42
pervasive computing, 1
Petri nets, 132
PHENOM, 1
play, 39
    action, 42
    actor, 42, 113
    content element, 42
    director, 42, 113
    mapping, 40
    performance, 42
    script, 42
    timing, 40
Plugin, 92
position, 278
power distance, 190
Presence, 174
presence, 174
    Ecological Validity, 174
    Engagement, 174
    Negative Effects, 174
    Spatial Presence, 174
Presentation, 359
Pride, 174
PriorityQueue, 328
Proactor, 72, 275
progression directions, 279
Proxy, 272
ProxyPullConsumer, 320
ProxyPullSupplier, 317
ProxyPushConsumer, 319
ProxyPushSupplier, 315
ProxyStreamSupplier, 345
PullConsumer, 320
PullSupplier, 316
PushConsumer, 318
PushSupplier, 315
PushSupplierImpl, 352
Put, 267

QoSProperties, 323

Rapid robotic prototyping, 24, 237
RDF, 46
Reactor, 71, 298
ReadyAction, 302

Real-time Channel, 88
  DataInterest, 325
  Dispatcher, 331
  Filter, 324
  PriorityQueue, 328
  QoSProperties, 323
  RTChannel, 334
  RTConsumerAdmin, 336
  RTProxyPushSupplier, 336
  Scheduler, 327
  TypedData, 324
repeatCount, 378
repeatDur, 378
repeating positions, 278
restart, 377
reverse tracking, 371
role type, 40
RTChannel, 334
RTConsumerAdmin, 336
RTProxyPushSupplier, 336

Scheduler, 271, 327
script, 39, 42
script time, 119
seq, 374
SGML, 29
SMIL, 27, 43
Spatial Presence, 174
spiral model, 5
StartAction, 302
state, 277
StateSyncEvent, 280
StateSyncEventHandler, 293
storyline, 29
StoryML, 28
  actor, 30
  dialog, 29
  environment, 30
  feed-forward, 29
  feedback, 29
  media objects, 30
  storyline, 29
Stream, 339
StreamAdmin, 342
StreamConsumer, 346
StreamConsumerAdmin, 344
StreamConsumerImpl, 366, 367
Streaming Channel, 93
  DataCache, 340
  DataSink, 340
  DataSource, 341
  DataSourceReceiver, 341
  ProxyStreamSupplier, 345
  Stream, 339
  StreamAdmin, 342
  StreamConsumer, 346
  StreamConsumerAdmin, 344
  StreamingChannel, 341
  StreamSupplier, 345
  StreamSupplierAdmin, 343, 347
StreamingChannel, 341
StreamSupplier, 345
StreamSupplierAdmin, 343, 347
StreamSupplierImpl, 366
SupplierAdmin, 312
syncbase, 373
SyncEvent, 280
SyncFactoryImpl, 305
Synchronizable, 280
Synchronizable Object, 69
  ActiveEventDispatcher, 299
  ActiveSynchronizable, 288
  ActiveSyncMediaObject, 290
  Event, 279
  EventDispatcher, 298
  EventHandler, 299
  MediaObject, 289
  StateSyncEvent, 280
  StateSyncEventHandler, 293
  SyncEvent, 280
  Synchronizable, 280
  SyncMediaObject, 288
  TimedMediaObject, 296
  TimedSyncEvent, 292
  TimedSyncEventHandler, 293
  TimedSynchronizable, 292
  Timer, 290
synchronization states, 277
synchronized media and interaction, 18
SyncMediaObject, 288
syncronization coordinates, 277
SyncService, 301

SyncServiceFactoryImpl, 307
SyncServiceProxy, 302
syntactic sugar, 133, 202
synthetic temporal relation, 118

tangible interaction, 14
temporal point net, 125
TentativeResult, 275
theater, 39
time-dependent element, 118
time-independent element, 118
Timed Action, 65
    Action, 267
    ActionService, 265
    ActionServiceProxy, 272
    ExecutionQueue, 268
    Scheduler, 271
    TentativeResult, 275
    TimedAction, 270
TimedAction, 270
TimedMediaObject, 296
TimedSyncEvent, 292
TimedSyncEventHandler, 293
TimedSynchronizable, 292
timeline, 124
Timer, 290
timing, 39, 40
TOONS, 12
TypedData, 324

ubiquitous communication, 2
ubiquitous computing, 1, 2
uncertainty avoidance, 190
user experience, 171

View, 354

Wrapper, 266

XML, 29

JFS
J.F. Schouten School for
User-System Interaction Research

TU/e industrial design