UNIVERSITÄT DORTMUND
Fachbereich Informatik
Lehrstuhl für Software-Technologie

M E M O Nr. 136

Vitruv: Specifying Temporal Aspects of Multimedia Presentations — A Transformational Approach based on Intervals

Klaus Alfert

April 2003

Internes Memorandum des
Lehrstuhls für Software-Technologie
Prof. Dr. Ernst-Erich Doberkat
Fachbereich Informatik
Universität Dortmund
Baroper Straße 301

D-44227 Dortmund

ISSN 0933-7725

Vitruv: Specifying Temporal Aspects of Multimedia Presentations — A Transformational Approach based on Intervals

Dissertation

zur Erlangung des Grades eines

DOKTORS DER NATURWISSENSCHAFTEN

der Universität Dortmund
am Fachbereich Informatik

von
Klaus Alfert

Dortmund

10. Oktober 2002

Tag der mündlichen Prüfung:	14.02.2003
Dekan:	Prof. Dr. Bernhard Steffen
Gutachter:	Prof. Dr. Ernst-Erich Doberkat
	Prof. Dr. Volker Gruhn

Contents

1. Introduction 1
1.1. The Altenberg Cathedral Project 1
1.1.1. Overview of the Project 1
1.1.2. The Software Engineering Dimension 3
1.1.3. Our Approach in the Altenberg Cathedral Project and its Problems 7
1.1.4. Summary and Problem Statement 9
1.2. Our Solution Proposal: Vitruv 11
1.3. How Vitruv works 12
1.4. Discussion and Related Work 14
1.4.1. Using Natural Language 15
1.4.2. Prototyping and Participative Design 17
1.4.3. Modeling Vague or Imprecise Requirements 18
1.4.4. Conclusion 19
1.5. Overview 19
1.6. Acknowledgements 20

I. Setting 21

2. Basic Terms 23

3. Related Work 27
3.1. Formal Models 27
3.1.1. Conceptual Models 28
3.1.2. Logical Models 29
3.1.3. Operational and other Models 33
3.2. Specification Techniques 35
3.2.1. Petri Nets 36
3.2.2. Statecharts 37
3.2.3. UML 37
3.3. Software Engineering Approaches 38

3.3.1. Requirements Engineering 39
3.3.2. Software Engineering based Hypermedia and Multimedia Design Methods 41
3.4. Assessment and Conclusion 44
3.4.1. Formal Models 45
3.4.2. Specification Techniques 46
3.4.3. Software Engineering based Approaches 46
3.4.4. Conclusion 47

II. Defining Vitruv 49

4. An Introduction to the Vitruv Approach 51
4.1. Basic Concepts of Vitruv 51
4.2. Vitruv_L 52
4.2.1. Language Features 52
4.2.2. Conceptual Modeling and its Realization 53
4.3. The Semantics of Vitruv_L 53
4.3.1. Static Semantics 53
4.3.2. Dynamic Semantics for Event-Free Behavior 54
4.3.3. Dynamic Semantics for Event-Based Behavior 54
4.4. Vitruv_N 55
4.4.1. Language Features 55
4.4.2. Example 56
4.5. Sketching a Process Model for Vitruv 56

5. The specification language Vitruv_L 59
5.1. Overview 59
5.2. Intervals and their Relationships 60
5.2.1. Intervals 60
5.2.2. Interval Relationships 61
5.3. Classes Structuring the Specification 62
5.3.1. Classes 62
5.3.2. Inheritance 65
5.4. Fuzzy Types 67
5.4.1. Structure and Properties of Fuzzy Types 67
5.4.2. The Standard Type DURATION 69
5.5. Events, Loops and Branching 70
5.5.1. Events 70
5.5.2. Loops 72
5.5.3. Branching 73
5.5.4. Multiple Events and Multiple Reactions 76

5.6. Presentations and Scenes 78
5.7. The Prelude 79
5.8. The Binding 81
5.8.1. L-Values 81
5.8.2. Defining Fuzzy Sets 81
5.8.3. Defining Modifiers 82
5.8.4. Contexts 83
5.8.5. Compound Interval Relations 84
5.8.6. Fuzzy Types 84
5.8.7. Binding of Classes 86
5.8.8. Binding of the Prelude 91
5.8.9. Scenes 93
5.8.10. The Main Entry Point 93

6. Static Semantics of Vitruv_L 95
6.1. Introduction 95
6.2. Preliminaries 97
6.2.1. Identifiers and Numerals 97
6.2.2. Type Attributes 97
6.2.3. General Building Blocks 99
6.3. The Entire Specification 102
6.4. Fuzzy Types 103
6.5. Example of a Type Derivation 105
6.6. Compound Interval Relationships 107
6.7. Classes 109
6.7.1. Representing of a Class 110
6.7.2. Sub-Classing and Inheritance 111
6.7.3. Exports 111
6.7.4. Local Declarations 112
6.7.5. The Class Definition 113
6.8. The Binding Section 119
6.8.1. Representing the Binding 119
6.8.2. The Entire Binding 121
6.8.3. Objects, Classes and Types 122
6.8.4. Assignment of Values 125
6.9. Remarks 128

7. Vitruv_I, the Intermediate Language for Vitruv_L 131
7.1. Event-Free Behavior 131
7.2. Language Description 132
7.3. Semantics of Vitruv_I 135
7.3.1. Abstract Syntax of Vitruv_I 135

7.3.2. Semantical Objects 135
7.3.3. Operational Semantics 137
7.4. Linearizing Vitruv_L 142
7.4.1. Fuzzy Types 143
7.4.2. Compound Relations 144
7.4.3. Classes 144
7.4.4. The Prelude 146
7.4.5. Blocks: Dealing with Loops, Selectors and Scenes 146

8. Vitruvian Nets 149
8.1. Prelimaries: Abstract Vitruvian Nets 149
8.2. Vitruvian Nets with Fuzzy Timing 153
8.2.1. Formal Definition 153
8.2.2. Translating Interval Relationships of Vitruv_L to FTVN 159
8.3. Vitruvian Nets with Fuzzy Markings 163
8.3.1. Translating Selections in Vitruv_L to Fuzzy Vitruvian Nets 163
8.3.2. Formal Definition 165
8.4. Nets for Events, Selectors and Loops 168
8.4.1. Basic Vitruvian Nets 169
8.4.2. Event Subnets 172
8.4.3. Selector Subnets 174
8.4.4. Loop Subnets 176
8.5. Nets for Scenes 178
8.5.1. Vitruvian Nets 178
8.5.2. Scenes and Leaving Them 181
8.6. Composition of Scenes: Translating Vitruv_L 183
8.6.1. Auxiliary Functions 184
8.6.2. Connecting Scenes 186
8.6.3. Expanding Events 188
8.6.4. Expanding Loops 189
8.6.5. Expanding Selectors 190
8.6.6. Net Construction 191

9. Specifying with Natural Language 195
9.1. Basic Considerations 195
9.2. Language Elements 197
9.2.1. Preliminaries 197
9.2.2. Presentation and Scenes 198
9.2.3. Media Definition 199
9.2.4. Media Composition 200
9.2.5. Omitted Elements of Vitruv_L 206
9.3. Mapping Vitruv_N to Vitruv_L 207

9.3.1. Preliminaries . . . 207
9.3.2. Presentations and Scenes . . . 208
9.3.3. Media Definitions . . . 208
9.3.4. Media Composition . . . 210
9.4. Assessment of Vitruv_N . . . 213

III. Applying Vitruv 217

10. The Multimedia Cathedral 219
10.1. Scene Collection . . . 219
10.2. Specifying with Vitruv_N . . . 220
10.2.1. The Intro . . . 220
10.2.2. Main Menu . . . 221
10.2.3. Various Cathedrals . . . 222
10.3. Cathedrals in Vitruv_L . . . 224
10.3.1. The Introduction Scene . . . 225
10.3.2. The Main Menu . . . 229
10.3.3. The Various Cathedrals . . . 232
10.4. The Cathedrals as Vitruvian Net . . . 238
10.4.1. The Presentation Structure . . . 238
10.4.2. Scene MainMenu . . . 240
10.5. Summary . . . 248

IV. Summary and Future Work 251

11. Summary 253

12. Directions of Future Research 255
12.1. Usability and Tools . . . 255
12.2. Language Extensions . . . 256
12.2.1. Fuzzy Types . . . 256
12.2.2. Connections between Scenes . . . 257
12.3. Generalizations . . . 258
12.3.1. Multimedia Complete . . . 258
12.3.2. Beyond Multimedia . . . 259

13. Final Remarks 261

V. Appendices 263

A. Definition of Vitruv$_L$ 265
A.1. Concrete Syntax 265
A.1.1. Preliminaries 265
A.1.2. The Structure in General 267
A.1.3. Fuzzy Types 267
A.1.4. Compound Interval Relationships 268
A.1.5. Events 268
A.1.6. Classes 268
A.1.7. The Binding 270
A.2. Standard Modifiers 273
A.3. The Standard Prelude of Vitruv$_L$ 273

B. Some Definitions and Results from Fuzzy Set Theory and Petri Nets 279
B.1. Fuzzy Set Theory 279
B.1.1. Fuzzy sets 279
B.1.2. Fuzzy Logic 284
B.1.3. Fuzzy Relations 284
B.1.4. The Extension Principle 285
B.1.5. Fuzzy Numbers and Intervals 286
B.1.6. Possibility Theory 287
B.2. Petri Nets 288
B.2.1. Multi-Sets 289
B.2.2. Petri Nets 290

Bibliography 293

List of Figures

1.1. Exploring the Altenberg Cathedral 2
1.2. Altenberg Cathedral: Discussing the Clerestory. 4
1.3. The models of Vitruv 15

3.1. The seven interval relations of Allen (1983). 31

5.1. The temporal arrangement of a, b, c, d, e and i. 72
5.2. The terms of fuzzy type Brightness (spec. 5.12) 85

6.1. Syntax of type attributes 98

8.1. The Hierarchy of Vitruvian Nets 150
8.2. Simple P/T-net with arc-weights as abstract VN. 153
8.3. Allen's Relations as Fuzzy Timing Vitruvian Nets 160
8.4. The body of class Example as FTVN 162
8.5. Comparison of a value with two alternatives. 164
8.6. BVN for Events. 174
8.7. The net for selectors. 175
8.8. The net for loops. 177
8.9. Two scenes with the connection place. 182
8.10. The Algorithm for Connecting Nets 193

10.1. UML Class Diagram for the Scene "Intro" 226
10.2. UML Class Diagram for the Scene "Main Menu" 230
10.3. UML Class Diagram for the Scene "Various Cathedral" 233
10.4. The basic structure of the presentation 239
10.5. The unexpanded body of scene MainMenu 245
10.6. The unexpanded body of loop waitingForTheEnd 245
10.7. The unexpanded path selTCS_body of selector selTCS 246
10.8. The scene MainMenu 247
10.9. The Loop-Body of waitingForTheEnd 249

B.1. The s-, z- and π-functions 282

List of Short Specifications

4.1. A simple example scene 57

5.1. Definition of the compound relation shortly after. 61
5.2. A simple class. 65
5.3. The definition of the class Interval. 67
5.4. The fuzzy type Brightness. 69
5.5. The fuzzy type DURATION. 69
5.6. Declaration of event pressed in class Button. 71
5.7. Loop until the button is pressed. 73
5.8. Select between additional video and audio depending on button event. 75
5.9. Multiple reactions to the button event. 77
5.10. Scenes and moving between them. 80
5.11. A small part of the standard prelude. 80
5.12. Binding of fuzzy type Brightness. 84
5.13. The demo classes A, B, C and D. 87
5.14. Assigning the value of large from DURATION. 88
5.15. Assigning an explicite fuzzy set expression. 88
5.16. Re-binding DURATION. 89
5.17. Binding a hierarchy of objects 89
5.18. Binding of an object with inheritance 90
5.19. Polymorphic Binding of an Object 91
5.20. Binding of Loops and Selectors 92
5.21. Binding with main entry point and scenes. 94

7.1. A simple Vitruv$_l$ example 134
7.2. Fuzzy Type Brightness in Vitruv$_l$ 143
7.3. Re-binding term black in Vitruv$_l$ 144
7.4. Allocation phase for assigning large to the length of a. 145
7.5. Application phase for assigning large to the length of a. 147
7.6. Declaring blocks in Vitruv$_l$ 148

9.1. Scene Definitions 199
9.2. Media definition 200

9.3. Content Elements 200
9.4. Media composition without events 202
9.5. Branching to another scene 205
9.6. Loops in $\mathsf{Vitruv}_{\mathsf{N}}$ 206
9.7. The first scene in $\mathsf{Vitruv}_{\mathsf{L}}$ 209
9.8. Media definitions in $\mathsf{Vitruv}_{\mathsf{L}}$ 210
9.9. Media definitions with elements in $\mathsf{Vitruv}_{\mathsf{L}}$ 211
9.10. Media composition without events in $\mathsf{Vitruv}_{\mathsf{L}}$ 213
9.11. Branching to another scene in $\mathsf{Vitruv}_{\mathsf{L}}$ 214

List of Long Specifications

10.1. The Code-Frame for the Cathedrals Presentation 224
10.2. The Introduction Scene 227
10.3. The Main-Menu Scene 231
10.4. Scene Various Cathedral 234
10.5. Scene MainMenu in $\mathsf{Vitruv}_{\mathsf{I}}$ 241
A.1. The Standard Prelude of $\mathsf{Vitruv}_{\mathsf{L}}$ 274

1. Introduction

The development of large multimedia applications reveals similar problems to those of developing large software systems. This is not surprising, as multimedia applications are a special kind of software systems. Our experience within the Altenberg Cathedral Project showed, however, that during developing multimedia applications particular problems arise, which do not appear during traditional software development. This is the starting point of the research reported in this thesis.

In this introduction, we start with a report on the Altenberg Cathedral Project (sec. 1.1), resulting in a problem statement and a list of requirements for possible solutions. After that we propose our solution named Vitruv (sec. 1.2 on page 11) and explain how it works in general (sec. 1.3 on page 12). It is followed by a discussion of key aspects of Vitruv and relations to other approaches (sec. 1.4 on page 14). The introduction closes with a brief outline of the thesis.

1.1. The Altenberg Cathedral Project

In the Altenberg Cathedral Project, the Chair for Software Engineering and the Chair for History of Architecture, both at the University of Dortmund, have worked together since 1996, aiming at a multimedia teaching system presenting an example for Gothic architecture. As the building of interest we chose the Altenberg Cathedral, a well-known Cistercian monastery in the Rhineland, containing all features needed for the intended system.

In the next section we present the project and its scope in greater detail. This is followed by a discussion of software engineering aspects.

1.1.1. Overview of the Project

Teaching the history of art is traditionally done without computers, especially without multimedia technology. Despite its centuries-old traditions, one might consider the history of art to be a promising candidate for a multimedia approach, since it consists of a large body of objects needing visual presentation. But this point of view neglects the discipline's scholarly aspects which are quite similar to those of other humanities. A multimedia system supporting teaching of the history of art has to combine both, visual presentation and scholarly aspects, to offer a real benefit compared to traditional text books or lectures.

The approach taken in the Altenberg Cathedral Project is twofold, consisting of a rich media environment and of scholarly discussions.

In the first part, we benefit from a multimedia environment, because we use many more media objects, with smaller cost, than in a traditional setting. This additional set of media objects makes it possible to visualize the topics discussed in greater detail, making the didactical point clearer. The potential to use different kinds of media, including video and audio, further enriches the learning environment. A particular set of media objects deals with a virtual reality-like model of the building. The virtual model of the cathedral allows users to explore the building freely and gives them more insight into the forms and views of the building, as compared to those few pictures usually presented in textbooks. Additionally, the virtuality of the building allows the observer to visit areas and viewpoints usually not accessible in reality. This makes it even possible to explore the building in its different stages of construction during the previous centuries. The observer can compare modifications made to the building throughout its development.

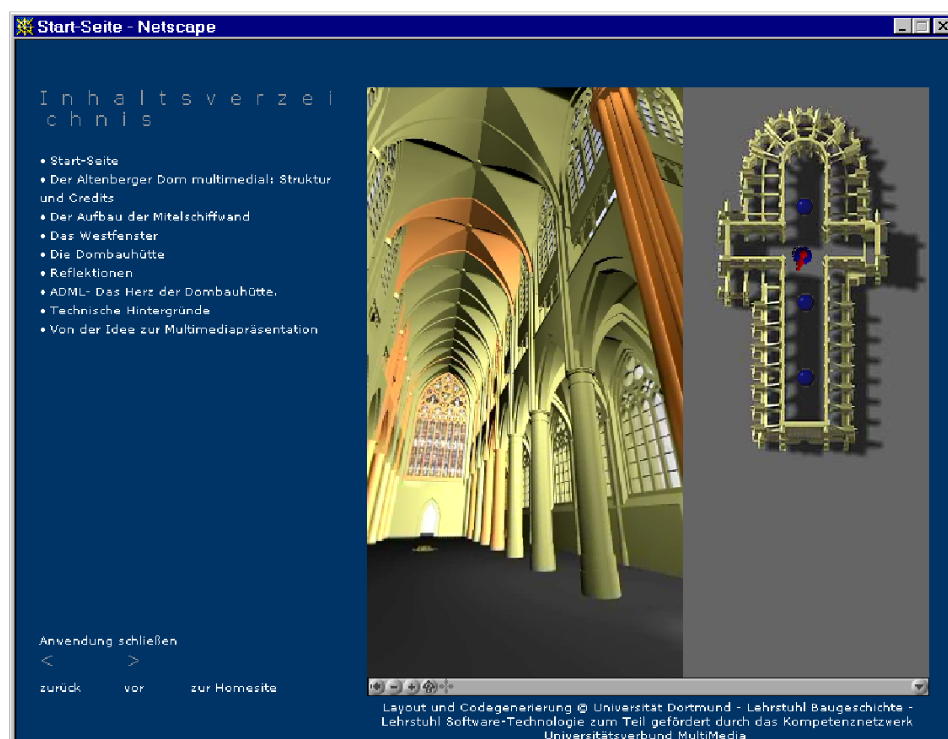

Figure 1.1.: Exploring the Altenberg Cathedral

In the second part of our approach, traditional scholarly discussions are presented in addition to the extended use of media objects. Many references in these discussions are linked directly to other ideas or media objects which further explain the topic. The

transfer of knowledge is obviously the primary system task subordinating the use of media: we have no interest in using media merely for the visual effect.

We have two ways of using the system: exploration and guidance. Our system starts with the presentation of the virtual cathedral. Users can freely explore the church from a set of different viewpoints familiarizing themselves with the building. “Exploring” means here that the user can turn around, choose between different viewpoints, zoom in and out on different details. In fig. 1.1 on the facing page we show the virtual cathedral. The user is positioned in the crossing[1] and looks to the west window, exploring the nave and the aisles. Some parts of the nave (and also of other areas of the virtual church) are sensitive to mouse clicks and are anchors of links leading to pages discussing the chosen topic in greater detail.

These pages complement the primary visual exploration of the church with scholarly discussions enhanced with media objects visualizing the discussed aspects. Approximately forty different topics are presented with support of a few thousand media objects ranging from simple pictures to video and audio clips, including also virtual reality scenes. In fig. 1.2 on the next page we show a discussion part of the structure of nave wall, here, in particular, the clerestory of predecessors of the Gothic style. In the picture on the left we see the interior of St. Appolinaris in Ravenna. Highlighted is the mosaic band above the arcades, an early form of the triforium.

This first way of using the system, the exploration, is supplemented by a second, arranging the topics and subtopics in a way independent of a specific part of the church, giving broader and more abstract topics a chance. It is essentially a set of guided tours through the material.

1.1.2. The Software Engineering Dimension

The Altenberg Cathedral Project is a large application developed and in use for teaching history of architecture for years. As such, it badly needs software engineering support during development and maintenance because of its size and complexity. In the following, we show that the development situation is different from other more ordinary software projects, and thus requires particular care.

1.1.2.1. Document-centric Multimedia Systems

Similar to ordinary text books, novels, movies, etc., multimedia applications aim at teaching or entertaining the user. Because of this, multimedia applications need the same careful construction as the classical media mentioned. Careful selection and arrangement of arguments, examples, presentation styles, etc. are very important and also very content-specific. From this perspective, multimedia applications have the

[1] For a good introduction to the architectural terms used we refer to the glossary on the web-site of Stones (1997).

Figure 1.2.: Altenberg Cathedral: Discussing the Clerestory.

same requirements and characteristics as traditional documents. We call this perspective of multimedia applications *document-centric*. The physical manifestation of the document-centric perspective are the media objects used, such as images, texts, videos, etc., and their arrangement. We call this manifestation the *document part* of a multimedia application.

document part

In addition to the document-centric perspective there also exists a strong program perspective in multimedia applications from which one part realizes technology itself, i.e., driver programs etc., whereas the second more interesting part supports the presentation logic. As such, the latter is tightly coupled with the document part and is called the *program part* of the multimedia application. Normally, programs or rather algorithms abstract from the processed data instances, they work on classes of these instances. Data instances in our context are the document parts mentioned previously. As these are very specific and also unique, they are often the only instance of their corresponding class. These singleton data sets couple the program part directly with the one and only document instance, intertwining program structure with document aspects. This makes it difficult to reuse one of both parts without the corresponding other part of the multimedia systems.

program part

In general it is not desirable to have such a tangle consisting of one program and one document. Data-driven programs are one approach of generalizing such programs towards a greater class of possible data instances they can work on. Markup languages like HTML or LaTeX show the power of this approach for more ordinary documents. But analyzed carefully, a dependency between these markup languages and their application domain is observed. As an example, LaTeX is excellent for writing scientific papers but it is poor for poster work. Another working example is the visualization of database contents such as electronic product catalogs. The structure of the database classifies the objects presented and their visualization. Whereas a catalog is primarily a simple list of items, other presentations are more complex with a corresponding complexity of their data structure. Ultimately, such presentation systems lead to data structures interpreted as algorithms and programs itself. This approach is a well-known technique in artificial intelligence and is usually found in LISP programs (Norvig, 1992). A LISP function is a special kind of a list, the only (native) data structure in LISP. The interpreter approach of LISP allows the construction of programs creating lists (i.e., data), which in turn can be interpreted as functions by the apply-function. This construction shows prominently that data structures can evolve to algorithms and programs.

Currently, we find a similar development in the WWW consisting of HTML files enriched with JavaScript. These documents have to be interpreted and executed by the browser to reveal their information. This situation leads the data-driven program approach *ad absurdum*. The genericity of these programs is complemented by the complexity of the used data which is specific to the presented document. It makes no difference whether we have a specific presentation program for each document, or a specific complex data-set for each document encoding presentation program parts.

The situation becomes awkward when changes in the document part also demand changes in the program and vice versa. The interconnection between both parts is complex, it is hard to assess the amount of changes in one part required by changes in the other part.

1.1.2.2. The New Developer Roles

technical developer

non-technical developer

As the two different parts of multimedia applications suggest, we have also two different groups of developers. On the one hand we have the usual *technical developer* for program development, which is often an academic, well educated in computer science, mathematics, engineering or natural sciences. On the other hand, we have the *non-technical developer* working on the document part. The group of non-technical developers includes writers, composers and musicians, artists, illustrators, movie people and others, often complementing the group of technical developers with respect to their primary education. In general, the group of non-technical developers is divided into the two subgroups of content and of media specialists.

In the Altenberg Cathedral Project we observed many misunderstandings between the project partners during requirements elicitation based on different knowledge and culture backgrounds. It was quite surprising to experience how substantial the differences between computer science and humanities really are regarding knowledge acquisition, intellectual tools, objects of research, etc. Therefore, we required a mutual understanding of our working fields and a common language. It is clear that the different backgrounds emphasize the need for such common languages – in a more homogenous setting it is not that obvious.

During traditional software development, e.g. for information systems, technical and non-technical people cooperate only at a few distinct time-points: during analysis and requirements elicitation, and during the final test stages. In our project, we observed that technical and non-technical developers work together throughout the entire development process. This was in part due to an iterative process with prototyping where analyzing, developing and testing were repeated many times, hence the aforementioned time-points occur repeatedly, too. But more important than that was what happened during each iteration. The dualistic nature of multimedia applications with its document and programs parts couples both developer groups more tightly. In multimedia applications similar to the Altenberg Cathedral Project the document part is nearly completely fixed after the delivery, hence the document part is created parallel to system development. The effect is that structural changes of program parts often require also changes in the documents parts and *vice versa*. This situation is quite contrary to ordinary information systems, where usually merely the structure of the database is fixed during development and only users create documents after system installation.

Additionally, we also had some elements of participative design, where the non-technical developers prototyped new user interaction mechanisms. Hence, computer

science technologies needed to be applicable for non-technical developers, which was only in part feasible. The lack of technical training of the arts scholars made it very difficult for them to use programming environments including advanced HTML editors with scripting facilities. We also observed difficulties in discussing the requirements after prototyping: without a commonly understood language for describing the dynamic behavior it is arduous to validate such requirements.

The added document part of multimedia applications makes the difference compared to ordinary software products. The document production should be considered explicitly in development processes. Thereby, the new non-technical roles become more important and a smooth integration of their creative work into the process is favored including the possibility to communicate easily between the heterogenous developer groups.

1.1.2.3. The Technical Side

On the technical side, we observe very rapid technological changes in the field of multimedia technology. The pace of hardware development influences directly the software's potential to manage more data in less amount of time and hardware consumption. The commercially available development tools are eager to support these new possibilities and do this by neglecting backward compatibility to some extent. In this way the rapid appearance of technology generations results in an ongoing development effort, reconstructing the entire application within in each new generation of development tools. Apparently the loss of maintainability seems to be no problem to the producers of such tools. The document character of multimedia applications seems to justify their opinion, as documents are quite seldom modified after their final production and publication. Traditionally, different editions of documents exist only for textbooks and encyclopedias.

In the Altenberg Cathedral Project, we have just such a combination of textbook and encyclopedia, together with a timeless topic, since the Gothic architecture will also most likely be taught in the future. Thus, we need here to abstract as far as possible from both current technologies and development tools to achieve maintainability and thus improve our investment in intellectual work.

1.1.3. Our Approach in the Altenberg Cathedral Project and its Problems

As related above, the Altenberg Cathedral Project consists of a collection of scholarly discussions. We were able to model the historian's didactical concept in a static structure as one of our main results during the requirements engineering phase. This static structure gave birth to ADML, the “Altenberger Dom Markup Language”, a specification language and an instance of XML (Bray et al., 1998). By the way, the Altenberg

Cathedral is the only church we know which has a specification language named after it.

Our art historians, the authors, write their texts in ADML and specify the use of media objects and the linkage between different parts of the document. Afterwards, a compiler is used to generate code for a set of different multimedia platforms, thus building the target multimedia system (Alfert et al., 1999). Currently, we support versions for Macromedia Director and HTML. All knowledge of the different technology platforms is hidden within the compiler and is not visible to the authors. As only the compiler has to be modified to support the technology changes mentioned above, the authors are not required to revise their original work.

Our authors, the art historians, dislike ADML in its raw form because of its formal syntax (and thus technical nature). This is the reason for us to search for better solutions. A conceptually simple approach is to construct an editing environment for ADML hiding the formal syntax. Another, more challenging problem with ADML is its focus on static structure and on scholarly discussion enriched with media objects. This works for the current state of the project, but ADML fails, when it comes to complex visualizations including user interaction. In this situation the smooth combination of document and program aspects becomes more and more important, as the tight integration of media objects, rendering control and user interaction handling is needed to get a working visualization.

The technical challenge of such multimedia presentations is the *synchronization of media in time (and space)* with user interaction, since many media objects occupy time and space (we come back to this issue in greater detail in chapter 2, Basic Terms). Specifying the behavior of media objects in time and space introduces the area of concurrent and real-time systems. It is well known that constructing concurrent and real-time systems is tricky and error-prone, and usually avoided if possible. Fortunately, in the multimedia domain hard real-time constraints are often not needed and we are in a more relaxed situation: following Little (1994) it is not harmful e.g. if we have to drop a few frames in a video. Nevertheless, the complexity of describing the synchronization of media objects is reflected by the large amount of proposals in the literature for specifying temporal behaviors (e.g. those discussed in chapter 3: Little and Ghafoor (1990); Diaz and Sénac (1994); Hardman et al. (1994); Khalfallah and Karmouch (1995); Vazirgiannis et al. (1996); Al-Salqan and Chang (1996); Zhou and Murata (1998); Paulo et al. (1999)).

Thus, we need specifications and abstract descriptions for media objects, rendering control and user interaction, i.e. these descriptions have to deal with the synchronization of media objects. We require these specifications and abstract descriptions in the case of such complex visualizations to gain both, well understood and complete requirements. A pragmatic solution in the spirit of ADML would also demand independence from current technology.

Let us consider as an example a presentation showing different French Gothic cathedrals, e.g. only those of Chartres and Amiens for the sake of brevity. We can

consider the following situation:

Example 1.1 (French Cathedrals)

> The presentation starts with Chartres, followed by Amiens. During the presentation of the cathedral of Chartres we hear an audio-clip explaining the cathedral. An information button appears during this presentation, a click on it links to another scene explaining the cathedral of Chartres in greater detail.

□

The presentation of Amiens should be structured similar. The example exhibits a set of implicit and explicit synchronization constraints such as the appearance and disappearance of user interface elements depending on the progress of the video presentation of both churches, i.e. depending on the content of this video. We also recognize a non-sequential ordering as we may branch to the more detailed presentation. But this branching is not always possible. As the use of not formalized natural language increases the risk of ambiguity, it is clear that a non-ambiguous specification of this example is highly desirable. But it is not clear how and in which formalism we can express this presentation.

Thus, let us recall the situation discussed earlier. We identified new development activities and developer roles for the development of multimedia presentations. The new developers have an educational background different from the usual technical developers, especially there might be a typical lack of technical and mathematical knowledge. This differs from the usual development situation where such differently educated people work together only at distinct phases of development, namely in the requirements engineering and the final test stages. But now we have to cope with this heterogenous blend of people throughout the entire development process. Common understanding is required, but tools and languages used during development make heavy use of technical and mathematical concepts. So, such language tools cannot form the basis for such a common understanding between both developer groups.

1.1.4. Summary and Problem Statement

While analyzing the development of the Altenberg Cathedral Project we have observed a new kind of developer, the non-technical developer, also appearing together with new roles serving the planning and the production of multimedia material. Producing this material is the document part of development, which is at least as important as the software part. Additionally, we observed a tight coupling between both parts influencing each other in different ways. Traditional software processes need to be adapted to support this new situation. It is especially important to establish a mutual understanding between technical and non-technical developers, because this is the heart of development processes for multimedia. Furthermore, elaborated

multimedia presentations have complex dynamic behaviors, requiring careful specification which has to be understandable to all involved developers and technically precise enough to support system construction.

One could argue that experiences made in the Altenberg Cathedral Project are not transferable to the development of other multimedia applications, since a project team consisting of scholars and computer scientists is quite unusual. But as Bailey et al. (2001b) report, similar problems arise in (commercial) multimedia agencies. They present a study of multimedia designers and their practices, revealing that there is a severe communication problem between (graphical) designers, programmers and customers, in particular with respect to the dynamic behavior of multimedia presentations.

Concluding, we can summarize our analysis and thoughts to the following problem statement:

> We require a specification language for multimedia presentations, which is simultaneously commonly understood and technically precise, such that technical and non-technical developers can use the language.

The approach we are looking for has to satisfy certain requirements discussed above and found more or less implicitly in the problem statement. To make easier references to the requirements, we list them here explicitly again. The first three are important to get the approach to work as a whole. The fourth requirement defines a weaker condition.

Requirement 1 (Understandability) *We need a commonly understood language for heterogeneous teams of developers. The language should not explicitly make use of mathematical concepts.*

Requirement 2 (Specification) *We need a specification language that is technically concise enough to allow system construction in an unambiguous way.*

Requirement 3 (Synchronization) *We need to describe the temporal behavior of media objects related to other media objects. This includes also user interaction insofar it controls the behavior.*

These three requirements are linked together such that the commonly understood language is a specification language for multimedia presentations with means for synchronization.

Efficiency during development is an important issue for larger systems. Efficiency is mainly supported through tools. Such tools may vary widely, from simulation to prototypes consisting of executable specifications, from theorem proving to model checking, from (graphical) syntax-oriented editors to compilers. The wide range of

possible tools makes it difficult to compare the tools and their corresponding approaches. Nevertheless we prefer approaches where tool support is possible and state it as a weaker requirement, our fourth.

Requirement 4 (Tool Support) *We prefer an approach with tool support to ease large scale development.*

1.2. Our Solution Proposal: Vitruv

Vitruv

To satisfy the aforementioned requirements we propose in this thesis the *Vitruv* approach. The central idea of Vitruv is to use natural language (NL) as common notation. Apparently, NL is commonly understood even for heterogenous developer groups, as demanded in req. 1 on the facing page. With NL, we can specify multimedia presentations, as indicated in example 1.1 on page 9, where we described a situation inside a multimedia presentation. In Vitruv, we focus on specifying temporal and behavioral aspects of multimedia presentations, according to requirements 2 and 3.

NL is often used in requirements engineering as common base for communication between developers and clients (Kotonya and Sommerville, 1998, p. 19), but usually with the well-known unpleasant taste of imprecision, ambiguity, vagueness and incompleteness, which are considered a remarkable risk. Therefore, we take particular care of these risks in Vitruv. Additionally, this care is important for satisfying requirement 4, because tool support demands a well-defined semantics, and for satisfying requirement 2, asking for unambiguous and concise information.

In this thesis, we provide the definition of Vitruv, such that development of tools can start. Empirical studies, answering questions concerning pragmatics and usability for complex projects, can be undertaken after the advent of appropriate tools and thus are left to future research (see sec. 12 on page 255).

We named our approach after Vitruv, an antique author on Roman architecture. He gained importance with his treatise *De architectura* in the Renaissance as source towards the antique arts and architecture by giving artists such as Leon Battista Alberti or Piero della Francesca the ideal of the proportions of the human body as base for their own art work (Fleming and Watkin, 1999). In this thesis, we pick up this idea and try a similar movement from technical-based specifications towards more human-centered specifications.

We should mention that at Carnegie-Mellon University, Mary Shaw, David Garlan and others have participated in a project regarding software architecture also named after Vitruv since at least 1995. They refer to Vitruv because of his influence in architecture and civil engineering in general, which is a different point of view towards Vitruv's work and its reception.

1.3. How Vitruv works

For Vitruv, we decided to use NL as common base for the communication between the heterogenous developer groups. We name this part of the Vitruv approach *Vitruv$_N$*.

Vitruv$_N$

To be more precise, we use only a restricted subset of NL for Vitruv$_N$. This has several advantages as we explain in the next paragraphs. Nevertheless, we preserve core features of NL, in particular good readability and understandability for both developer groups is important here. One could argue that we reinvent a language feature as existing for forty years in COBOL, since COBOL is somewhat talkative (or noisy) to support non-technical readers (Horowitz, 1984, p. 15). However, COBOL-like languages – and even more conventional languages such as Java – have a (grammatical) structure, which is not similar to longer NL texts at all: these languages mimic only simple sentences. In contrast to that, in Vitruv$_N$ we focus on preserving the look and feel of NL. Therefore, we stick to the notion of a NL-based language for Vitruv$_N$ although Vitruv$_N$ uses only a subset of NL.

The decision to use NL as basis of a specification language demands that we deal with the inherent problems of NL, namely with ambiguity, imprecision, vagueness and incompleteness. In the following, we present how we tackle these problems.

Ambiguity is hard to resolve completely. This is due to its various sources in NL such as multiple meaning of words or unclear references of pronouns. We diminish ambiguity in Vitruv$_N$ by providing a careful selected subset of NL, where both, grammar and vocabulary are restricted. Of course, this reduces the generality, which NL otherwise would provide, but since we focus on specifying multimedia presentations, we do not consider the loss of generality as harmful. Difficulties in writing specifications in Vitruv$_N$, which arise from the lesser degree of freedom, are outweighed by the better understandability due to reduced possibilities for ambiguities. Compared to an unrestricted use of NL, these restrictions ease the construction of supporting tools, because for instance the restricted grammar allows to use standard parsing techniques from compilers instead of general rewrite systems.

Detecting incompleteness of a NL specification is often tedious. It is more convenient to check for incompleteness in a formal setting. There, we have the need to be explicit and can not rely on implicit assumptions, and we gain the possibility to check formally whether the specification is consistent or complete. Of course, it depends on the formal calculus used how much a formal analysis can reveal.

To achieve a precise technical description we have to deal with the inherent impreciseness and vagueness in NL. With fuzzy set theory, introduced by Zadeh (1965) as a precise formal model of imprecision, we can adequately model imprecise and vague statements in the specification. But as for the problem of incompleteness, we are in need of a formal model. Nevertheless, of importance is that the imprecision is transfered into the formal model without the need to give up the imprecision and thus becoming overly (and easily arbitrarily) precise. Hence, the imprecision and vagueness in the NL specification is reflected also into the formal model.

These considerations suggest to introduce a formal counterpart to Vitruv_N, which shall make it possible to apply effectively fuzzy set theory and formal analysis to Vitruv_N specifications of multimedia presentations, i.e. we are in need for a formal semantics of Vitruv_N. Formal semantics are also required for tool support. But since the step from natural language to formal semantics is quite large and hence the mapping between these two concepts is complicated, we take an intermediate step and introduce the formal specification language *Vitruv_L*. We regard Vitruv_L as a mediator between Vitruv_N and its proper formal semantics. This is similar to the use of the programming language C as an abstraction of machine language, found in some compilers for high-level languages, where as first step the program in the high-level language is compiled down to a C program. Examples for such compilers are the ProSet compiler or the first compiler for C++ (Stroustrup, 1991, p. 6). To achieve native executables, in a second step the C code has to be compiled by a standard C compiler. Analogously, in our case, Vitruv_L connects specifications in Vitruv_N to their formal semantics and we first map Vitruv_N to Vitruv_L and in a second step Vitruv_L to its formal semantics. Because of the restricted subset of Vitruv_N and its anticipation of fuzzy set theory, it is possible to translate systematically a Vitruv_N specification to Vitruv_L; this is an important characteristic of Vitruv_L. Therefore, we are confident that the corresponding Vitruv_L specification stays close to the meaning of the original Vitruv_N specification. This is also needed for the way back transferring the formal statements about Vitruv_L to the context Vitruv_N, such that formal results can be presented to the non-technical developers in an appropriate form.

Vitruv_L

Compared to Vitruv_N, we add to Vitruv_L more technically required ballast, due to the need of being explicit as discussed above. This includes algorithmic details, the provision of a standard library of media types with their capabilites and a rich type structure with object-oriented concepts and statical typing. Of course, we make use of fuzzy set theory in Vitruv_L as well. The definition of the temporal structure is based on an extended version of the interval calculus of Allen (1983) (see sec. 3.1.2.2 on page 30 for details on the interval calculus). The extended interval calculus incorporates quantitative statements and nondeterministic events[2], for specifying interval durations and for modeling user input, respectively. We do not assume that non-technical developers work with Vitruv_L, but it is intended as a handy tool for technical developers for technically precise specifications of multimedia presentations, without loosing the ability to relate the formal specification to the NL specification.

The semantics of Vitruv_L is divided into three parts, following the "separation of concerns" principle of software design (Ghezzi et al., 1991). We distinguish between static semantics, and dynamic semantics for event-free and event-based behaviors. These three parts are handled separately, but are themselves not unrelated. If the static semantics or the dynamic semantics of the event-free behavior of a specification

[2]Nondeterministic means here that we have no control whether an event occurs or which value it might have.

is inconsistent, then also the dynamic semantics of the event-free or the event-based behavior are inconsistent, respectively. We present now the three parts in greater detail.

The static semantics of Vitruv_L is given as a deduction system in the tradition of formal type systems (cf. Cardelli, 1997). With the static semantics we can check the static typing of Vitruv_L, thereby preventing many specification errors in advance.

The dynamic semantics of the event-free behavior of Vitruv_L is concerned with the dynamic parts of a Vitruv_L specification which do not depend on nondeterministic events. To make Vitruv_L amenable to consistency checks of the event-free semantics, we have to identify and extract the respective parts of the Vitruv_L specification. This is the reason to introduce a simpler form of Vitruv_L, which lacks the rich type structure and the event handling. This simpler form has a similar task as Vitruv_L has for Vitruv_N, since it connects Vitruv_L and its semantics. Therefore, we regard this simpler form of Vitruv_L as an intermediate language and call it *Vitruv_I*. On basis of the operational semantics of Vitruv_I, we can apply consistency checking algorithms.

Vitruv_I

Finally, the entire dynamic semantics including the event-based parts are addressed by *Vitruvian Nets*, which are a Petri net variant. The use of Petri nets for modeling the dynamics of multimedia presentations has a long tradition. But more important is that the interval calculus and the events of Vitruv_L translate properly to Petri nets. In addition to that we enrich Petri nets with fuzzy set theory to model the imprecision and vagueness of Vitruv_L specifications, which themselves reflect these properties of the respective Vitruv_N specification. Thereby, we can model all important characteristics of Vitruv_L with Petri nets.

Vitruvian Nets

In fig. 1.3 on the facing page we summarize the relationships between the parts of Vitruv. On top, we have Vitruv_N, which is used by both, technical and non-technical developers. The semantics of Vitruv_N are given by Vitruv_L, which may also be used by technical developers for technical specifications. The three different parts of the semantics of Vitruv_L are shown in the dashed rectangle, where we identify the static semantics as formal type system, and the dynamic semantics based on Vitruv_I and Vitruvian Nets for the event-free and event-based parts, respectively.

1.4. Discussion and Related Work

The central ideas of the Vitruv approach, namely using NL as common notation between technical and non-technical developers and thereby allowing both developer groups to work jointly, touch aspects of requirements engineering approaches, since in the requirements engineering process technical and non-technical people work together on eliciting and analyzing the customer's requirements of the system to be built.

In this section, we discuss how Vitruv is related in general to requirements engineering approaches with respect to the use of NL (sec. 1.4.1 on the next page) and

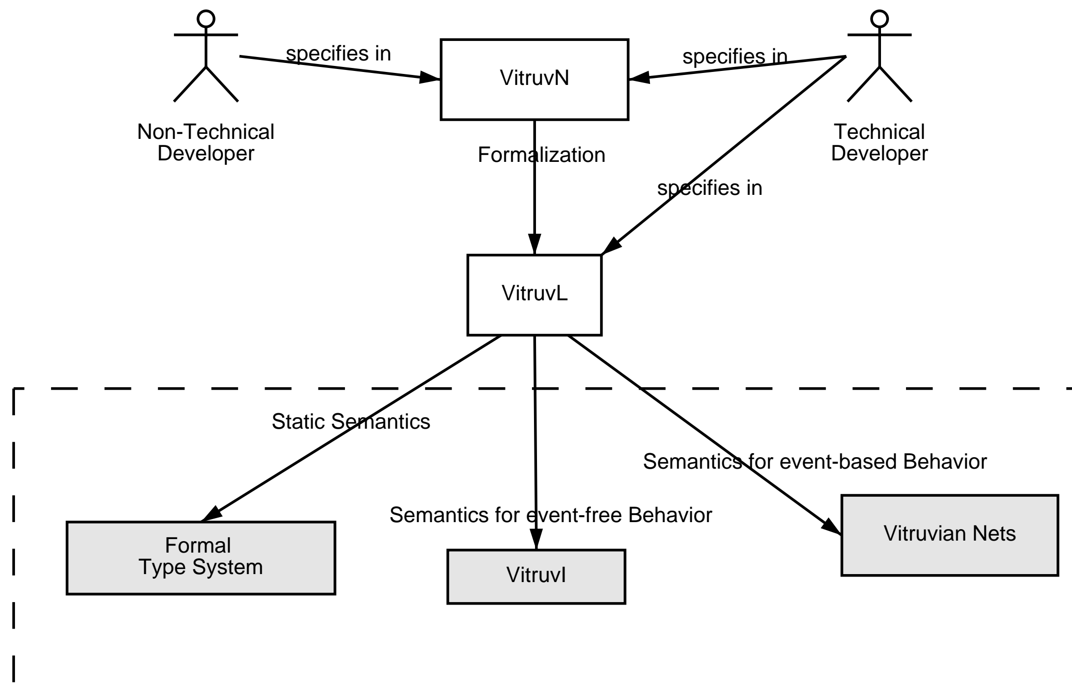

Figure 1.3.: The models of Vitruv

the incorporation of end users (i.e. non-technical people) in the development process (sec. 1.4.2 on page 17). Finally, in sec. 1.4.3 on page 18 we are concerned with modeling vague or imprecise requirements.

1.4.1. Using Natural Language

In Vitruv, we use NL as common base for understanding between heterogenous developer groups. This is similar to the well-established use of NL in requirements engineering (Kotonya and Sommerville, 1998), where the heterogeneity exists between developers and clients. In requirements engineering, NL is used for requirements specification documents, development contracts and other documents, which have to be understandable for both groups. Often, this use of NL appears also when using a semi-formal language for documenting requirements specifications. A typical situation is found in the context of UML (OMG, 2001), which provides for the analysis phase the diagram type for use cases. Methods applying UML (e.g. Fowler and Scott (1999, sec. 3) or Booch et al. (1999, p. 224)) suggest to specify use cases primarily with NL texts, which are annotated with UML's use case diagrams, thereby underlining the importance of NL for the early phases of software engineering. Summarizing, we consider the use of NL as a suitable and well-established practice.

However, the difference between Vitruv and other approaches is how we proceed with the NL specifications. Traditionally, it is the task of the requirements engineer to transform the NL requirements specification by hand into more formal calculi, including semi-formal approaches such as UML. The aforementioned disadvantages of NL (imprecision, ambiguity, vagueness and incompleteness) are remarkable risks concerning the translation process and make it difficult to ensure that the formalized specification meets the clients' expectations. Since clients in general have not enough knowledge about the calculus used, they cannot validate the formal model. Therefore, validation has to wait until the product is constructed and product tests begin. In Vitruv, things are different from that.

For Vitruv, we propose a systematic translation process from the NL specification in Vitruv_N to the formal specification in Vitruv_L. The systematic translation is possible because of the features of Vitruv. Vitruv_N uses only a restricted set of NL, where both, grammar and vocabulary are fixed. This eases the translation process and reduces the risk of ambiguity in Vitruv_N specifications compared to the free use of NL. On the formal side of Vitruv, we handle vagueness and imprecision with fuzzy set theory. Thereby, we translate vague or imprecise statements found in Vitruv_N immediately to Vitruv_L (and its semantics as well). No additional representations for the formalization nor inadequate ad-hoc defuzzyfications (cf. sec. 1.4.3) are needed. The immediate mapping of NL features into the formal world of Vitruv ensures that we stay close to the intended meaning of the NL specification.

In Vitruv, we focus on the systematic translation of NL specifications into formal specifications, but we require that the translation is done automatically. There are, however, other approaches, which try to derive automatically formal models from a NL specification. Some approaches (e.g. Gervasi and Nuseibeh, 2002) are lightweight formal methods, which often perform only a partial analysis of the entire specification. They elicit only few but important information from the specification and build a respective formal model. Whether the aforementioned risks occur depends on the specification documents used. Gervasi and Nuseibeh (2002) operate on technical specification documents from NASA and are interested in detailed specification of a system bus. Therefore, risks of ambiguity, vagueness and imprecision are reduced by the domain's very nature. Nevertheless, such approaches have the inherent disadvantage that it remains unclear whether they can elicit automatically all important and required information from their partial view at the NL specification. On the other side of the spectrum of approaches dealing automatically with NL specifications, are those which try to understand the entire specification by applying NL understanding technologies. As an example, the approach of Rolland and Proix (1992) derives ER models from NL specifications by identifying entities and their relationships in the NL text. As validation of their ER model, they generate a NL text from the ER model, which is to be checked by the customer. It remains open how these approaches deal with imprecision and vagueness of NL specifications, except that model and NL specification do not match in the validation. In sec. 1.4.3 on page 18 we come back to the

aspect of imprecision and vagueness but in a more general setting.

1.4.2. Prototyping and Participative Design

Beside the use of NL discussed above, prototyping and participative design (PD) offer other ways for incorporating non-technicians into to the development process.

Prototyping (Budde et al., 1992; Doberkat and Fox, 1989) aims at short development times between system generations, such that user feedback can easily be integrated in the next system generation. Evolutionary prototyping and in particular mock-up prototypes of user interfaces are well suited for studying and presenting the behavior and the look-and-feel of applications, such that non-technicians can validate the prototype with respect to their (sometimes implicit) requirements. The strong point of prototyping is that the technical development is parallel to requirements elicitation and stabilization, which is done jointly by technical and non-technical developers.

Prototyping is, however, very costly and thus seldom used in the commercial multimedia domain, as Bailey et al. (2001b) report. They argue that storyboards are more appropriate artefacts in early steps of the development process. Storyboards allow layout and content sketches, and outlines of the story line. They have the disadvantage that they are static, which makes it difficult to communicate behavior properly. Thus, designers often enhance storyboards with textual annotations describing the (temporal) behavior. Of course, these annotations use natural language and no formal notation, since designers are not comfortable with formal methods for similar reasons as the scholars in the Altenberg Cathedral Project. To solve this problem, Bailey et al. (2001a) propose a storyboard tool which allows animated behavior sketches by a visual language.

Participative design (PD; see e.g. the special issue of CACM, introduced by Kahn and Muller, 1993) is related to prototyping and aims at incorporating users of software products into the design process. Of course, technical aspects and details are not of concern, but of importance are overall functionality and user-interfaces. Often, PD is used as part of usability studies and for validating that the system to be built fits properly into the working place of the users participating in the design. Therefore, PD uses its own non-technical abstractions for a constructive design.

Both, prototyping and PD, show that detailed cooperation between technical developers and non-technicians provides benefits for system development with respect to user satisfaction. However, the situation in the application domain of Vitruv is different from these two approaches, such that we cannot simply adopt their techniques. This is due to the clear separation between developers and end-users, separating also the techniques used by these two groups: technical development on the one hand and evaluation, requirements statements etc. on the other hand. In the domain of multimedia applications, the separation between the tasks of technicians and non-technicians is much more blurred and cannot always be separated, resulting in the

distinction between technical and non-technical developers instead of developers and end-users in traditional settings. Therefore, we are in need for appropriate communication means, languages and tools for these two developer groups, which neither prototyping nor PD approaches provide.

1.4.3. Modeling Vague or Imprecise Requirements

Vague or imprecise specifications occur not only in NL specifications or in multimedia, but also in traditional specifications for software systems. In this section we take a look at approaches from the literature and relate them to the approach of using fuzzy set theory for modeling vague or imprecise requirements proposed here. We focus on timing considerations, since they are conceptually close to the important aspect of synchronization in multimedia applications.

Timing considerations are important properties of systems and applications, and have to be identified during the requirements elicitation phase. Usually, timing considerations are regarded as non-functional requirements (Sommerville, 1990; Ghezzi et al., 1991; Partsch, 1998; Kotonya and Sommerville, 1998). Whereas functional requirements are mostly subject to tests in the final product, non-functional requirements have the drawback that they are mostly not testable. In this situation, non-functional requirements degrade to wishes and guidelines similar to user interface guidelines providing opportunities for different interpretations.

Sommerville (1990) argues that sometimes it is possible to reformulate such not testable requirements by explicitly quantifying them: The specification "the system's response should be fast" can be changed to "the system's response should be in 2 seconds". Clearly, this is testable and we can build appropriate test scenarios e.g. for the system under various load levels. While this seems as if the aforementioned problems are solved, this is not really the case. The quantified requirement is still expressed in natural language, usually to achieve a better understanding of the requirements by the customers (Partsch, 1998, p. 21). However, this makes it dangerous to interpret this as quantification in a strictly mathematical sense. For instance, it is questionable whether a test case in which the system requires 2.01 seconds to response does not fulfill the requirement while another one needing only 1.99 seconds does. Often, we need to coarsen the precise value of 2.0 to some kind of a broader interval, since a quantification of "fast" resulting in the value of "$\leq$ 2 seconds" does not mean precisely 2. One classical approach to deal with this uncertainty is to allow some statistical error, such as to state that "in 95% of all situations, the system's response should be in 2 seconds". This approach has the advantage that its formal underpinning is well known by probability theory. It is, however, uncertain whether the introduction of such statistical errors was the intention of stating the system's response should be "fast".

A different approach to deal with vagueness and imprecision is *possibility theory*

(Dubois and Prade, 1999; Biewer, 1997) based on fuzzy set theory, which is only rarely used in requirements engineering, and particularly in modeling, with the notable exception of the work of Liu and Yen (1996), discussed later in sec. 3.3.1 on page 39. With possibility theory, we can model the quantification of "fast" as a set of possible values together with a grade of the possibility of the values. In some way, possibility theory is a generalized variant of probability theory (cf. sec. B.1.6 on page 287), since a possible value is not required to be very probable, however, every probable value has to be possible. The concept of linguistic variables in fuzzy set theory allows introducing a type "speed" with predefined imprecise and vague values such as "fast" or "slow", which are realized as fuzzy sets. Expressions with linguistic variables are formulated with these predefined values, hence the vagueness and impreciseness remains and it is not absorbed by some reduction to a single precise value. In fuzzy control theory (Yager and Filev, 1994) these concepts are applied successfully to model complex control processes even for which no classical analytical models exist. A particular interesting point is that expressions with linguistic variables are quite easy to understand independent of their complex non-linear formalization. This is a clear benefit for maintenance of such systems.

1.4.4. Conclusion

In Vitruv, we emphasize the systematic transformation of the natural language (NL) specification into a formal one, which takes particular care of imprecision and vagueness already existing in the NL specification. Other approaches deriving models from NL do not focus on that topic specifically. Alternative approaches for getting non-technicians involved into the development, we discussed participative design and prototyping, provide no appropriate means for technical precise specifications.

The formal model used in Vitruv uses fuzzy set theory for modeling vagueness and imprecision. The discussion on modeling vague and imprecise requirements above suggests that an approach guided by fuzzy set theory allows formal models which are close to the meaning of the original imprecise and vague requirements of the NL specification. This is exactly what we are looking for.

1.5. Overview

This thesis is divided into four parts, followed by appendices: In Part I, Setting, we discuss the setting of this thesis. We start with a definition of basic terms (sec. 2), followed by an analysis of related work with respect to the requirements of Vitruv (sec. 3).

In Part II, Defining Vitruv, we present our approach in detail. We start with a short outline of the entire approach in sec. 4. We define Vitruv_L informally in sec. 5, followed

by the static semantics in sec. 6. In sec. 7 we present the semantics for the event-free behavior. We linearize Vitruv_L by defining Vitruv_I, an intermediate language for Vitruv_L. The definition of Vitruvian Nets follows in sec. 8. We close this part with presenting Vitruv_N in sec. 9.

Part III, Applying Vitruv, presents a larger example, showing how the various parts of Vitruv interact and how they are related (sec. 10). Part IV, Summary and Future Work, closes the main part of the thesis. In sec. 11 we give a summary, followed by a discussion of future research directions (sec. 12).

The final Part V, Appendices, collects two appendices. The concrete syntax, the standard modifiers and the prelude of Vitruv_L are given in app. A. Finally, we present basic definitions and results of fuzzy set theory and Petri nets (app. B). The thesis closes with a bibliography and an index.

1.6. Acknowledgements

I would like to thank my supervisor Prof. Dr. Ernst-Erich Doberkat for providing the environment and the freedom for doing research on Vitruv, as well as his support and encouragement, constructive discussions and critics throughout the time are appreciated. Prof. Dr. Volker Gruhn provided constructive critics which helped to clarify and to focus the presentation.

I spent many hours together with Dr. Alexander Fronk, discussing hyper- and multimedia, formal methods and the quest for the meaning of all that. I would like to thank him also for the practical and moral support needed from time to time. I appreciate the discussions about the Petri nets and the helpful comments on earlier versions of this thesis by Uschi Wellen. My (former) students, Matthias Heiduck, Christoph Begall and Marc Störzel discussed and developed parts of Vitruv. The Altenberg Cathedral Project Team, Prof. Dr. Ernst-Erich Doberkat, Corina Kopka, Matthias Heiduck, Jens Schröder, Jens Scharnow, Prof. Dr. phil. Norbert Nussbaum, Dr. phil. Thorsten Scheer and Dr. phil. Stephan Hoppe, developed the initial starting point of this thesis, thereby also teaching the beauty of Gothic cathedrals. Dr. Malcolm Usher helped polishing my English, however, all remaining mistakes in this thesis are my very own and he is not to blame at all.

Part I.

Setting

2. Basic Terms

Before going on, we are in need for definitions of the rather vague terms concerning multimedia, multimedia presentations and applications, and hypermedia. They are given in detail in this section. We close the section with a discussion of the differences of multimedia and software engineering documents.

Multimedia Systems

Multimedia as a general term is used in public discussions as a synonym for any modern computer technology with direct end-user impact. Examples of these are personal computers, the Internet, sometimes even telecommunication products such as cellular phones, personal digital assistants (PDA), and of course digital media types such as digital video, MP3, DVDs, etc. In Germany, multimedia was the word of the year 1995[1], underlining the importance and broadness of this term in every-day life.

In a technical setting, multimedia is understood as a combination of different media types. In contrast to ordinary systems, especially graphical systems, in *multimedia systems* at least one of these different media types has to be time-dependent, such as audio, video or animations (Koegel Buford, 1994b, p. 2). Sometimes, systems handling only one time-dependent media type are also called multimedia systems. Examples for the latter are video-on-demand servers. The definition suggested here is more focused than those in public discussions, but its scope is also very broad: it ranges from network technology for transporting media to databases storing and querying media, from device drivers to multimedia presentation software. This interpretation of multimedia is usually applied in technical journals such as ACM Multimedia Systems or IEEE Transactions on Multimedia focusing on implementing rather than applying multimedia technology.

multimedia systems

It should be mentioned that time-dependency in multimedia systems is different from those found in, e.g., information systems with temporal data. In such systems datasets have a time tag indicating the temporal validity of the dataset, e.g. day and time for share prices in stock tickers or the date of entries in accounting systems. In contrast to that, time-dependency in multimedia systems means that each time-dependent atomic information unit has a time dimension and consequently an extension into time, as found, for instance, in digital video clips. Here, the temporal data is

[1] Each year selected by the *Gesellschaft für deutsche Sprache*, the list is available online at `http://www.gfds.de/woerter.html`.

no additional attribute but is rather one of the main characteristics of the information units.

Multimedia Applications

In the context of the present work, we do not concentrate on technical problems of implementing multimedia technology but rather on applying multimedia technology. Thus, our definition of multimedia focuses stronger on human perception, following the definition given by Newcomb et al. (1991) in their article on HyTime: *multimedia* is

multimedia

> *"a parcel of information intended for human perception that uses one or more media in addition to written words and graphics. The presentation of the added media may occupy time, space, or both".*

The important difference of this definition with respect to the aforementioned definition of multimedia systems is the explicit focus on human perception. It implies that e.g., the transportation of a combined set of different media objects including time dependent media by a broadband carrier is not considered as multimedia. Far more, it is important to prepare carefully the media objects to transmit the intended message, focusing on human communication. That is why we indicated in the description of the Altenberg Cathedral Project that multimedia comes with a strong document aspect. The preparation of multimedia data together with information about how to render this data is called a *multimedia document*.

multimedia document

multimedia presentation

A *multimedia presentation* is a piece of software presenting a multimedia document to the user. Similar to the well-known WWW browsers such as Netscape or Microsoft Internet Explorer, they allow primarily read-only access to the information and offer little possibilities to modify the appearance of the presented data or to change the data itself. Thus, multimedia presentations maintain the document character of multimedia, because the human perception is their main task. Presentation programs can be divided into two separate groups: generic and individual ones with respect to the presented material. WWW browsers are apparently generic programs as they are not specific to a document and can present nearly everything on the WWW. In contrast, the presentation program in the Altenberg Cathedral Project realizes the requirements specific to this very application and the documents created in the project.

Currently, we observe substantial additional scripting in HTML documents for achieving individual user interfaces and user interaction. This emphasizes the combination of document and program aspects in multimedia documents and blurs the separation of the two different groups of presentation programs. We focus in the context of this thesis on these multimedia presentations, that are tightly coupled to the corresponding multimedia documents supporting an individual presentation style for each data instance. This model encompasses also systems presenting large databases

(popular examples are web banking applications, digital libraries, or e-commerce portals), albeit these systems only work on classes of data and do not use individual presentation schemes on data instances.

Beside multimedia presentations there is a broader kind of multimedia systems called ***multimedia applications***. These applications differ from multimedia presentations because they do not require that their multimedia material remains constant and in general users of these applications can modify the multimedia material. Obviously, development tools such as Macromedia Director (Epstein, 1999) belong to this category. In this thesis, we do not deal with such applications but concentrate rather on multimedia presentations.

multimedia applications

Hypertext and Hypermedia

Conceptually close to multimedia are the terms hypertext and hypermedia. Some authors use them as synonyms, such that both, hypertext and hypermedia may mean the same but also that hypermedia and multimedia are essentially the same. In this thesis, we differentiate between all three terms, the difference, however, is subtle.

With ***hypertext*** we characterize a text enriched with associative references within the text, where the references link portions of the text. These relationships form a graph structure and usually provide an easy access to related parts of the text. Usually, the graph structure is not a simple list or tree-like, but is expected to be more complex. Since hypertext usually is supported by a computer system, the definitions for multimedia documents, applications and systems apply analogously for hypertext documents, applications and systems.

hypertext

For the difference between hypermedia and hypertext it is important that for hypertext we only consider pure text together with linking information. If we add media elements to a hypertext, e.g. images or videos, we talk about ***hypermedia***, i.e. hypermedia extends hypertext by additional media elements. Since these media elements may in particular occupy time and space, hypermedia may also be called multimedia, and, indeed, many problems in hypermedia and multimedia are the same. However, we differentiate between hypermedia and multimedia such that in hypermedia the focus lies on the linkage structure whereas in multimedia the media arrangement is of primary concern. This difference is subtle and a unique assignment can only be done for some extreme examples: multimedia presentations with a strict sequential order are usually not regarded as hypermedia, since there is no appropriate link structure; hypertexts with images are not regarded as multimedia, since media elements with temporal aspects are missing.

hypermedia

Therefore, the assignment of a specific document or presentation to either hypermedia or multimedia is rather arbitrary. It is possible to consider the same document or presentation as hypermedia or as multimedia, however, the viewpoint changes: in the first case, we accentuate the link structure between the various information pieces

presented, in the second case we emphasize the media arrangement presented to the user.

Multimedia vs. Software Engineering Documents

To complete the picture we should now differentiate between multimedia documents and classical software engineering documents. At first glance, it seems that the document part of a multimedia application is not different from other documents created during the development process of ordinary software systems, documents such as requirements and design documents or user documentation.

This observation is true up to a certain point. Obviously, multimedia documents have to be handled in a similar way as other documents, i.e. configuration management should be applied to them, reviews are required, etc. For short, all usual practices of project management should also be applied to the development of multimedia documents. Nevertheless there are important differences. First, multimedia documents contain always time-dependent media, which are rarely used in current software engineering documents. The production process of these media artefacts is different from traditional documentation. Second, multimedia documents are a central part of the final product, often they are the main part *per se*. In contrast to that, software engineering documents have the flavor of add-on products, the main product is the software developed, the documents are needed primarily for efficient (and effective) development and users' training. Third, the interconnection between multimedia documents and their software product is more complex than between a software product and its software engineering documents, because software engineering documents describe (or prescribe) facts about the software product, whereas multimedia documents contain parts of the software product, in particular the aforementioned rendering information. Finally we should notice that multimedia documents are primarily created by non-technical developers, while software engineering documents are primarily created by technical developers.

In summary, we conclude that multimedia documents and their creation process are significantly different from ordinary software engineering documents.

3. Related Work

For selecting techniques for in particular the formal models of Vitruv, we discuss now related work found in the literature. Much work has be done in the area of multimedia in general, but there is only little attention on specializing software engineering approaches to the multimedia domain. If we do not focus on the software engineering aspect only, we find several approaches addressing the development of multimedia applications which will we discuss in the following sections. The focus of these approaches lies mostly on formal models of the application's temporal behavior. Only a minor group of publications regards problems of the development process itself such as new process models, specific approaches for requirements engineering, specification or design languages and others.

We structure the presentation of related work such that each section focuses on of the first three requirements defined in sec. 1.1.4 on page 9. However, we reverse the order of the requirements in this presentation for didactical reasons, since we prefer to discuss foundations before their application. We start in sec. 3.1 with approaches dealing with synchronization aspects only, i.e. with requirement 3. After that, specification languages (requirement 2) are discussed in sec. 3.2 on page 35. These languages consider synchronization aspects, hence they also satisfy requirement 3. Finally, in sec. 3.3 on page 38 our discussion is concerned with aspects of understandability (requirement 1), presenting development models supporting the entire range of system development. Finally, we close the discussion with a conclusion (sec. 3.4 on page 44), where we evaluate the mentioned approaches and argue why we need a new and different approach for developing multimedia applications.

3.1. Formal Models

We mentioned earlier that multimedia applications require media delivery of time-dependent material. In short, this results in two new activities, media production and temporal media composition. It is clear that the media production process is not formalizable and thus not of interest here. But the temporal composition of media is formalizable similar to the composition of concurrent processes. Therefore, the foundation of temporal media composition is essentially the same as for concurrent processes.

We begin this section with an introduction of conceptual models of time. These models form the basis of formalization carried out in the models presented later. The

first set of these formal models consists of an adoption of formal specification methods for concurrent systems. The second set of models owes more to multimedia applications. These models are designed in a more pragmatic way. Finally, we present fuzzy temporal models.

3.1.1. Conceptual Models

Time is a phenomenon not directly perceptible for humans by sense. Thus, the notion of time is always bound to some indirect measures which are of course interpretable. Historically, from the antiquity to the middle-age, measurement of time was based on events such as high noon or sun rise. The time between such events was divided into a set of intervals ("hours") of which the absolute length depended on the current day-length: in summer, hours were longer than in winter. In the advent of the modern age, the development of mechanical clocks led to the independence of the length of hours from the current day-length.

But modern clocks do not change the two fundamental concepts or manifestations of time. Firstly, we have *point-like structures* imposed by the mentioned events. In particular the rather artificial event of a certain time of the day, e.g., three o'clock, is seen as a time-point without any extension. This point of view leads to the mathematical model using the real numbers for representing time as used in physics and engineering. Secondly, we have *temporal intervals* as every observable event has an extension. In natural language it is complicated to deal with time-points directly. Often, a kind of threshold is used to reduce a very short interval to a time-point. This situation indicates that time points are an abstract concept not directly observable.

Reasoning about time always deals with these two concepts. In physics and engineering, time is usually a one dimensional variable of real numbers. The use of calculus allows, e.g., to model the current speed v as the partial derivative of length x of a trajectory by time t:

$$v(t) = \frac{\partial x}{\partial t} = \dot{x} \Longleftrightarrow x = \int_0^t v(\tau)\, d\tau$$

These models were introduced 1687 in Isaac Newton's *Principia Mathematica* and have been successfully applied since them. The success of this model is founded at least on its decoupling from human every day reasoning and the elegance of its mathematical formalism. On the other hand, in physical experiments the interval concept is always visible as part of the measurement exactness of time: there is also a transition from temporal intervals to time-points without reaching the mathematical definition of a point without extension.

The interval approach is more appealing in a setting suitable for every day reasoning, a domain of Artificial Intelligence often based on logical theories. Reasoning about time is important for the areas of planning, natural language understanding

and others (Habel et al., 1993). Hajnicz (1996) distinguishes two fundamental concepts: facts and events. A fact is a logical statement describing the state of the modeled world. A fact is static and cannot be changed by other facts, but its value can change over time. In contrast to facts, an event models the dynamic view of the world: an event happens and changes by this the value of (some) facts. Both concepts together suggest to reason about temporal intervals while certain facts remain immutable. An example could be:

> Peter went by train from Dortmund to Düsseldorf. After that, he took the plane to Paris. He read a book during the flight.

The different kinds of travel are the facts which remain constant during their respective intervals. Other temporal intervals can be related to them, in our example it is the interval in which Peter is reading the book. In another way, the flight can be seen as an event changing the fact, that Peter travels with the train. This event clearly has an extension and is an interval, too. This model allows the division of one event into several sub-events occurring during its interval giving events an internal structure. In contrast to the real numbers model of time, the interval model suggests to reason in a qualitative and not always quantitative way, as we did not mention the amount of time needed for traveling.

Both models are important and form the basis of different formalizations and applications. Often they are used jointly since we have knowledge about quantitative properties of intervals and additionally some qualitative relations between intervals.

3.1.2. Logical Models

In the following we present logical models for time. First, we discuss temporal and modal logics, then we focus Allen's interval logic and finally we consider a fuzzy approach.

3.1.2.1. Temporal and Modal Logics

Modal logic (Hajnicz, 1996; Emerson, 1990) is a branch of non-classic logic and uses a set of possible worlds in which the formulas hold. The simplest modal system, K, is an extension of propositional logic consisting of a non-empty countable set P of propositions, a set of standard logical connectives $(\neg, \rightarrow)$, an unary modal operator $\Box$ called *necessary* and another modal operator $\Diamond = \neg\Box\neg$ called *possibly*. The semantics for modal logic is based on the *possible worlds semantics* of Kripke (Emerson, 1990, p. 1064). A model $M = (W, R, m)$ is a triple consisting of a set W of possible worlds, an access relation $R \subseteq W \times W$ and an interpretation m assigning to each proposition $p \in P$ the set of worlds in which p is satisfied. The logical connectives are interpreted in the usual way. The modal operator $\Box$ is interpreted as follows: $\Box\varphi$ is satisfied in a

world $w \in W$ if φ is satisfied in every world w' such that wRw'. The main axiom of modal logic is

$$\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B).$$

Together with the inference rule

$$\frac{A}{\Box A}$$

it constitutes the modal system *K*. More complex modal systems define axioms inducing properties of the access relation *R* such as reflexivity, transitivity or symmetry.

The main difference between classical and modal logic is the introduction of a set of possible worlds *W* and the relation *R* between them. This pair $\mathcal{K} = (W, R)$ is called a Kripke-frame. A very usual interpretation of Kripke-frames is that each world describes a state in a computation. The relationship between the worlds defines a path through this states. This interpretation allows to specify the behavior of programs. Historically, Pnueli (1977) was the first to use modal logic for specifying programs. The temporal logic of action (TLA) of Lamport (1994) is another well-known approach.

A different interpretation of Kripke-frames considers the sequence of worlds in the access relation *R* as sequence of time points, i.e., we interprete the access relation as time precedence relation (Hajnicz, 1996). The model consists of the triple $M = (T, <, v)$ with *T* as a non-empty set of time-points, the transitive asymmetric precedence relation $<$ between time-points, and *v* as valuation function assigning to each proposition *p* the set of time points in *T* where *p* is satisfied. The value of formula φ at moment *t* is written as $\varphi[t]$. The semantics are based on time structures $\mathcal{T} = (T, <)$ requiring transitivity and irreflexivity of the precedence relation $<$. It is important to notice that such time structures do enforce that the order of *T* is either discrete or continuous. Therefore, it is possible to define time structures isomorphic to $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ (Hajnicz, 1996).

Whereas modal and temporal logic are suitable mathematic models for formal reasoning about time, they are usually not applied in the multimedia domain. One important reason might be their abstractness which does certainly not appeal to non-mathematicians. Tool support is limited to quite simple logics, a prominent example is CTL (Computational Tree Logic, cf. Emerson (1990)), for which model checkers exist (Clarke et al., 1999). Without such tools, it is tedious and error-prone to use modal logics for constructive purposes because all necessary proofs need to be carried out by hand.

3.1.2.2. Interval Logic

Allen's seminal work on temporal interval logic (Allen, 1983) uses first-order predicate logic to model intervals and their relations. Allen defines thirteen qualitative relations between two intervals describing all possible positions. They are shown in

fig. 3.1. Allen's interval logic considers only intervals; points do not exist. Events and facts are related to intervals. His relations originate from every day reasoning and have an intuitive semantics even without formal definitions. For practical use, Allen presents a constraint solving algorithm with time complexity $\mathcal{O}(n^3)$ where n is the number of interval variables. The algorithm checks if a given set of intervals and relations between them is consistent, and calculates relations not explicitly given. This algorithm is an adoption of the Waltz algorithm for constraint solving (Norvig, 1992).

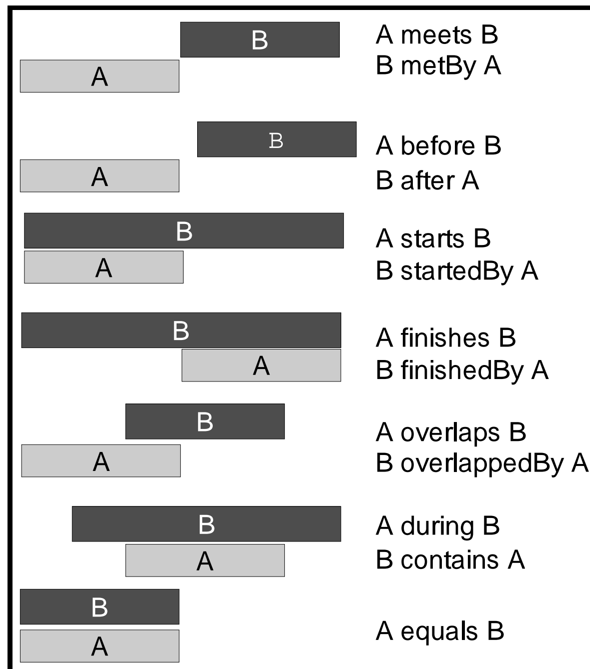

Figure 3.1.: The seven interval relations of Allen (1983).

A common notation is

$$\begin{array}{ccc} A & \{\text{meets}, \text{before}\} & B \\ B & \{\text{during}\} & C \end{array}$$

meaning

$$\begin{array}{ccc} ((A \,\text{meets}\, B) & \vee & (A \,\text{before}\, B)) \\ & \wedge & (B \,\text{during}\, C) \end{array}$$

The constraint solver calculates all missing relations. In our example, no relation is mentioned between A and C, which means that we have no information and each of the thirteen relations may hold. The constraint solver narrows this to

$$A \quad \{\text{during}, \text{before}, \text{meets}, \text{overlaps}, \text{starts}\} \quad C.$$

If the constraint solver calculates that no relation holds between two intervals, we have an inconsistency: the specification cannot be satisfied. However, Allen's algorithm considers consistency of three intervals only, for complete consistency between all n intervals we have a complexity with an exponential blow-up to $\mathcal{O}(n^n)$ (Vilain and Kautz, 1986).

As we can see, the constraint solver does not determine a unique solution but the set of possible relations. While this might be a useful situation for the planning domain (Allen's original working area), this is very often a problem for creating a schedule, which we also need in the multimedia domain (cf. sec. 2). A unique solution is usually required for a schedule.

Allen's work is not only widely recognized in artificial intelligence (Habel et al., 1993; Hajnicz, 1996), but also in the multimedia field starting with the work of Little and Ghafoor (1990) on Object Composition Petri Nets (OCPN). It is a good starting point for qualitative specifications, but as Goetze (1995) points out, the only relations that are exact in a way that they can prescribe precisely the temporal order, are the meets and equals relations. The knowledge of, e.g., A before B does not reveal anything about the interval between A and B except that its length is strictly positive. The lack of quantitative knowledge about intervals even decreases the possibility of specifying the temporal order.

Nevertheless, the work of Allen is important because of possible tool support and easily comprehensible relations. The problem of integrating quantitative knowledge is tackled by Dechter et al. (1991), and extended later by Jonsson and Bäckström (1998). Their work relaxes Goetze's criticism somewhat.

3.1.2.3. Fuzzy Temporal Models

Fuzzy temporal models deal with the representation of vague or incomplete knowledge about time. In contrast to classical approaches, where truth values are crisp, i.e. either 0 or 1, here any proposition concerning time may have a graded truth value.

The approach of Dubois and Prade (1989), embedding the former approach of Dutta (1988), is based on a continuous linear scale $\mathcal{T}$, modeling time. The knowledge about time-points (or dates) is represented by possibility distributions mapping from $\mathcal{T}$ to $[0, 1]$, restricting the more or less possible values for certain time-point. Let $\mathbf{a}$ be some time-point and $\pi_\mathbf{a}$ be the possibility distribution of $\mathbf{a}$, then $\forall t \in \mathcal{T} : \pi_\mathbf{a}(t)$ denotes the possibility that $\mathbf{a}$ is exactly t. The possibility distribution $\pi_\mathbf{a}$ is identified with a fuzzy set A with membership function μ_A of the more or less possible values of $\mathbf{a}$, i.e. the possibility degree of $\pi_\mathbf{a}(t)$ is exactly $\mu_A(t)$. As usual, a possibility degree for t of 0 means that t is not a possible value at all, a degree of 1 means that t is absolutely possible. If $\pi_\mathbf{a}(t_1) > \pi_\mathbf{a}(t_2)$, then we prefer t_1 to t_2, because the possibility of t_1 is of a higher degree than the possibility of t_2.

Dubois and Prade call $\mathbf{a}$ an *ill-known time-point*, because we have (usually) no certain knowledge about $\mathbf{a}$. Their model of fuzzy time-points assumes that a time-point

a has only one value. Therefore, all possible values, i.e. the members of the support of A ($\text{supp}(A) = \{t \in \mathcal{T} \mid \mu_A(t) > 0\}$), are mutually exclusive candidates for **a**. They call the (fuzzy) set A of possible values for **a** a *disjunctive set*. Additionally, Dubois and Prade require that the fuzzy set A for **a** is convex, all possible values for **a** are clustered. If A is also normalized (i.e. its core is not empty), then A denotes a fuzzy number or interval and we can apply fuzzy arithmetic operations, eg. for calculating the distance between two fuzzy time-points. In particular, it is possible to define fuzzy intervals between fuzzy time-points **a** and **b** as the intersection of the corresponding semi-intervals (cf. (B.42)–(B.45)). With fuzzy intervals, we can model fuzzy variants of Allen's interval logic, together with modified relations such as A slightly before B, imposing a length constraint on the interval between A and B.

Yager (1997) applied the approach of Dubois and Prade to multimedia information systems investigating fuzzy temporal queries on annotation-based video databases. The queries are of the form "who was the woman appearing on the horse about fifteen minutes into the video", where the fuzzy temporal expression of interest is "about fifteen minutes". However, there is no specific relation to multimedia at all, video information systems are only used to serve as an application domain, where temporal queries are useful. Nevertheless, Yager shows that fuzzy temporal models can be used in real world systems.

3.1.2.4. Summary

We presented logic based approaches for temporal behaviors. They all can be used to define the formal semantics of the temporal behavior, but their reception in the multimedia literature differs. We find virtually no reference for using temporal or modal logics. In contrast to that, Allen's interval logic is a standard model and has widely influenced multimedia research. An important reason for that is the temporal extension of time-occupying media for which intervals are an intuitive model. The fuzzy temporal approach has not reached a deeper impact yet.

3.1.3. Operational and other Models

Besides logical models we find as formal models also operational models, derived from the formalization of concurrent systems, and formal document models. These are presented now.

3.1.3.1. Formalization of Concurrent Systems

Research in the area of concurrent systems is focused on synchronization problems between processes running in parallel. The usual programming concepts for synchronizing parallel programs, such as semaphores, monitors, and message passing, are semantically equivalent (Tanenbaum, 1992). Formal models often reduce this situation

to programs with states and actions modifying the states, and the program's possibility to communicate with each other via channels. Prominent examples of such models, also called process algebras (Baeten and Verhoef, 1995), are CSP (Hoare, 1985) and Milner's CCS. There is a direct link between the sequence of states in these models and the relation of worlds in modal and temporal logics. Lamport (1994) defines the semantics of TLA via the sequence of states. Broy (1991) presents the sequence of states as the main elements of the formal model. Winskel and Nielson (1995) present various approaches in a unifying category theoretical model, including Milner's CCS, Hoare's CSP and even Petri Nets.

Few authors apply process algebras to model their multimedia systems semantically. Khalfallah and Karmouch (1995) designed a multimedia database and formalized the concurrent activities during rendering the multimedia information with CSP. In Goetze's Ph.D. thesis (1995), a multimedia system using the ET++ Multimedia Framework on the Unix operating system was developed. Goetze uses CSP for defining the semantics for rendering in his approach. Vazirgiannis established a similar system based on the Windows platform. He uses an algebraic approach to define the temporal behavior (Vazirgiannis, 1999; Vazirgiannis et al., 1996). A different approach is taken by Santos et al. (1999) using RT-LOTOS for specifying the temporal integrity of multimedia documents.

3.1.3.2. Document Models

Formal approaches devoted explicitly to multimedia usually regard the coding of multimedia data. This applies in particular to the development of media coding standards such as MPEG, JPEG, Quicktime or Video on Demand (eg. Steinmetz, 2000; Gibbs and Tsichritzis, 1995). Most of the time, aspects of application development with such media is not discussed.

This seems to be a bit different for hypermedia systems. Following the seminal Dexter reference model for hypertext (Halasz and Schwartz, 1994) formalized in Z (Spivey, 1992), some extension exist which deal with time based media. A direct successor of the Dexter model is the Amsterdam model (Hardman et al., 1994), a different approach is proposed by Tochtermann (1994). The task of these models is to formalize the common understanding of hypertext or hypermedia systems. This includes time based media for hypermedia systems, but the primary focus always lies on the link structure between certain information nodes. For instance, in the Amsterdam model information nodes can be composed of different parallel streams each modeling a temporal media object. These streams can then be used as link anchors. Despite their identification of components, abstractions and interfaces, the intention of all these models is not to define an architecture for the implementation of hypermedia systems. They deal with a model of hypermedia systems as a foundation of ongoing discussion: now it should be exactly clear what they are talking about discussing certain aspects of hypermedia systems.

A more constructive way is given by certain well-defined media standards. To mention are HyTime (Newcomb et al., 1991) and SMIL (Hoschka, 1998). HyTime is a SGML language and allows to define hypertext within a set of user-defined temporal coordinate systems. The newer SMIL language is based on XML (Bray et al., 1998) and is intended for streaming media in the WWW. SMIL consists of both, screen layout, using the well-known frame approach of HTML, and the specification of the temporal behavior of the different media rendered in those frames. The Z_YX model (Boll and Klas, 2001) is another XML based multimedia document model incorporating features from HyTime and SMIL. In contrast to aforementioned approaches, Z_YX has a strong focus on presentation reuse and adaption allowing for easy identifying, replacing and modifying of presentation parts. All three languages are useful and intended for implementing systems using explicit and precise temporal measures (milliseconds, SMPTE frames, etc.). However, this makes it difficult to use these languages during the early stages of system development where exact measures of media are usually not known.

3.1.3.3. Summary

The aforementioned formal operational and document models in the multimedia domain are either used for specifying the implementation semantics or for defining reference models, used as the basis of scientific discussions. In both cases, the application of formal models is similar to the use of semantical models for (sequential) programming languages. The primary use of, for instance, denotational or axiomatic semantics (Mosses, 1991; Winskel, 1993) is to define the precise meaning and interaction of language constructs. This allows to reason about the correctness of language constructs and programs, and serves as a formal specification for the implementation. All these properties hold also for the formal models for multimedia.

3.2. Specification Techniques

In real-time systems problems similar to the synchronization in multimedia applications have been investigated for several years resulting in a set of formal models and techniques, some of them already discussed in sec. 3.1.3 on page 33. Some of the developed techniques are also useful in other domains and so became part of the standard body of software engineering techniques. Of particular note are Petri nets and state transition diagrams which also are incorporated in simplified form in UML, the current *lingua franca* for object-oriented analysis and design (Fowler and Scott, 1999).

3.2.1. Petri Nets

All different kinds of Petri nets share their common structure as a bipartite directed graph consisting of transitions and places as different sets of nodes (a more formal introduction is given in appendix B.2 on page 288). Input and output places are those places connected with an edge from the place to the transition or from the transition to the place, respectively. Often places are interpreted as data storage and transitions as operations reading data from input places and writing to output places. This data is modeled as tokens, flowing through the net while consumed and produced by transitions. This is the dynamic behavior of a Petri net. The syntactic structure of the net defines all possible traces of tokens through the net. The behavior is then refined by defining the firing behavior of the net which states how and when a transition fires, i.e., how and when a transition consumes tokens and produces new ones.

Petri nets are organized hierarchically in classes (cf. Baumgarten, 1996; Reisig and Rozenberg, 1998). In elementary nets (also called event-condition nets), each token represents a boolean value stating that a certain condition is either fulfilled or not. Place-Transition nets (P/T-nets) enhance places to contain a set of tokens. Predicate-Transition nets (Pr/T-nets) allow to depend the firing of transitions on logical predicates. Colored Petri nets (CP-nets) introduce typing of tokens and places, and functions defining values of produced tokens (Jensen, 1997). The concept of hierarchical nets is applied to P/T- and CP-nets which allows modularization of specifications. Some variants of Petri nets consider time explicitly. They extend tokens with a timestamp. This timestamp effects the firing behavior because not only the sheer existence of tokens but also their timestamps decide whether a transition fires.

Usage of Petri nets for multimedia originates from Little and Ghafoor's work on their Object Composition Petri Nets (OCPN) in the early nineties (Little and Ghafoor, 1990, 1993). They based OCPN on Allen's interval calculus and strongly restricted OCPN's topology. The lack of both, cycles and non-determinism, in OCPN allows only simple pre-orchestrated multimedia applications where the presentation of the application is completely predetermined, and thus no user interaction is possible. Other approaches, e.g. Time Stream Petri Nets (TSPN) from Diaz and Sénac (1994) do not have this disadvantage. They use timed Petri nets to model complex firing behaviors including the ability to model user interaction. A quite different area is tackled by Al-Salqan and Chang (1996), using timed Petri nets for specifying the behavior of distributed multimedia systems. The consideration of additional jitter during media transport in networks is a major problem. But this is a technical problem of multimedia systems and multimedia applications as it concerns the enabling of multimedia technology in distributed environments.

Looney (1988) introduces a combination of Petri nets with fuzzy set theory as a generalization of elementary nets, where Boolean valued tokens are replaced by tokens representing fuzzy values. These Petri nets are called Fuzzy Petri nets and are used for maintaining knowledge in knowledge based systems. Several variants ex-

ists focusing on different aspects of Petri nets, Cardoso et al. (1996) give an overview. A rather different approach is presented by Murata (1996), where fuzziness is introduces in timed Petri nets. In contrast to the aforementioned approaches, the fuzziness appears only in the time-dependent parts of the Petri nets, i.e. tokens have a fuzzy timestamp, transitions fire with a fuzzy delay and the enabling and occurrence time of transitions is fuzzy. The marking itself – except for the timestamp – is independent of fuzzy values, in particular Murata uses a colored high-level net. This is, however, not relevant for the approach a fuzzy timed Petri nets. As an application in multimedia, Zhou and Murata (1998) discuss the use of fuzzy timed Petri nets for modeling distributed multimedia systems with respect to network delays and jitter, similar to the aforementioned work of Al-Salqan and Chang (1996).

3.2.2. Statecharts

Statecharts (Harel, 1987) are an extension of simple state-based automata such as Mealy- or Moore-automata. Statecharts allow the modeling of complex specifications by hierarchies. The states can be refined into a set of further states in which exactly one state always has to be active if the surrounding state is reached (XOR-state). The surrounding state is an abstraction of the refinement states. Additionally we have a decomposition of a state, in which all inner states have to be active in parallel (AND-state) modeling concurrent behavior. Both kinds of decomposition can be recursively applied. Transition between states are called events and can be annotated with conditions and actions which must hold or are executed, resp., while moving from one state to another.

Statecharts can be compiled into programs, their formal semantics are defined by Harel and Naamad (1996). Also usual analysis techniques such as reachability, deadlocks detection etc., are applicable to statecharts.

Paulo et al. (1999) present an extended variant of statecharts, called hypercharts, for modeling hypermedia applications. Their extension covers more complex synchronization behaviors designed after TSPN (cf. the last section) introducing timed transitions. Similar to high-level Petri nets, the formal semantics of hypercharts are expressed via ordinary statecharts.

3.2.3. UML

UML (OMG, 2001; Booch et al., 1999; Fowler and Scott, 1999) is currently the *lingua franca* for object-oriented analysis and design. It combines a set of static and dynamic diagram languages linked through a common meta-model (Rumbaugh et al., 1999). In fact, only the meta-model is standardized by the OMG, everything else is derived from that. A major problem of UML is the lack of a sound formal basis, as only the syntax of its meta-model if formally defined. While this gives enough freedom to use

UML as a communication vehicle during development and a suitable language for brainstorming, it is nearly impossible to apply formal analysis or code generation based on a common understanding. Certainly, UML tools exist especially for code generation but these tools apply their own semantics usually only implicitly defined in the code generators program code. Currently, UML is enriched with OCL, the Object Constraint Language, to annotate the diagrams with formally notated constraints (Warmer and Kleppe, 1999). It is unclear whether this movement leads towards a formally defined and widely accepted semantics of UML.

The dynamical models for concurrency are statecharts and activity diagrams. Statecharts are used for modeling different states and transition in one object or class, where activity diagrams model different concurrent threads of control outside of a class boundary. Activity diagrams are a variant of control flow diagrams with support for concurrent threads. They resemble Petri nets because of being a bipartite graph, where one set of nodes models activities and the other set synchronization points. Both models depend on UML's class diagrams and therefore do not have a proper formal semantics. Additionally, both models require a substantial training as discussed above in the section about statecharts.

The OMMMA approach (Sauer and Engels, 1999) is an extension of UML for modeling multimedia applications. OMMMA adds to UML a new diagram type addressing layout considerations and extend several other diagram types, e.g. sequence diagrams for concurrent activation of several media objects and state diagrams to react on spatial events. OMMMA addresses some missing properties of UML for the development of multimedia applications. Its technical point of view reveals that OMMMA focuses on the design phase and less on the analysis phase.

3.3. Software Engineering Approaches for Development of Multimedia Applications

Software engineering deals with all aspects of developing software with a strong focus on large systems. Several sub-disciplines have been developed focusing on e.g. database applications, real-time applications, parallel and distributed systems.

In order to avoid misunderstandings, we have to be very careful when talking about software engineering for multimedia. Let us distinguish again between multimedia systems in general and multimedia applications (cf. section 2). The latter deals with applying multimedia technology whereas the former provides multimedia technology. Software engineering support for providing multimedia technology (Mülhäuser and Effelsberg, 1996) is quite established and well understood, especially as the development environment is a purely technical one. We focus here on software engineering support for multimedia applications.

From the broad range of general software engineering methods and techniques,

we turn our attention towards two areas: requirements engineering (sec. 3.3.1), and hypermedia and multimedia design methods (sec. 3.3.2). We select these two areas since both are concerned with central aspects in this thesis, other areas of software engineering are not of our concern. While requirements engineering is not specialized towards multimedia, it focuses on building suitable technical models from non-technical descriptions and requirements, thereby dealing in particular with our Requirements 1 (Understandability) and 2 (Specification) in a more general setting. Hypermedia and multimedia design methods are devoted explicitly to the multimedia domain, thus, by their very definition, they should address our requirements.

3.3.1. Requirements Engineering

Requirements engineering, a sub-discipline of software engineering, is concerned with requirements elicitation, their negotiation between different stakeholders (e.g. managers, users or system administrators), their documentation and specification, and finally their verification and validation (cf. Pohl, 1996). All these activities involve cooperation between customers, users and requirements engineers. That is why understandability, our Requirement 1 (p. 10), is an important issue for requirements engineering: we need mutual understandability between all participants throughout the requirements engineering process, because otherwise customers, users and developers cannot agree on the same requirements.

Modern requirements engineering approaches apply a set of different models for the various groups of stakeholders, aiming at achieving a complete set of requirements from different viewpoints (Finkelstein et al., 1992; Pohl, 1996). Since these viewpoints usually reveal only a partial view of the entire system, it is crucial to unify them to achieve a more complete system description. The range of models used for specifying and documenting requirements is broad, from natural language documents via semi-formal languages such as UML to formal specification languages such as Z (Spivey, 1992); Partsch (1998) gives a good overview. This broad range makes it less easy to unify the models since they address different areas of interest with a different level of detail and a different vocabulary, resulting easily in a set of inconsistent specifications. Addressing these inconsistencies is an open research question, various approaches are proposed, e.g. using quasi-classical logics (Hunter and Nuseibeh, 1998). However, modeling the users' requirements into a logical formalism is a difficult task. For development of information systems, Rolland and Proix (1992) have done some research on automatically transforming natural language specifications into conceptual data models by means of linguistic analysis of the sentence structure. For error checking of the synthesized model they provide a translation back to natural language, which can be rechecked by domain experts. An alternative to this approach is to use light-weight formal methods, which do not attempt to formalize the entire specification but merely check the specification partially. Gervasi and Nuseibeh (2002)

present such a light-weight method, constructing the model from a natural language specification with domain-based parsing techniques (Gervasi, 2000). In contrast to the free use of natural language, Fantechi et al. (1994) present an approach for automatic translation of a restricted subset of natural language to action-based temporal logic (ACTL).

Requirement statements are often imprecise. A typical reason is that stakeholders stating the requirement simply do not know any better, but sometimes requirements cannot be stated precisely due to their very nature, e.g. statements about efficiency. Classical approaches to the formalization of requirements do not adequately tackle this imprecision, i.e. even with a formal language such as first-order predicate logic the imprecision often remains (Pohl, 1996, p. 24). A notable exception is the approach of Liu and Yen (1996) using fuzzy set theory for modeling vague requirements. Their approach is also useful for requirements negotiation between different stakeholders, since requirements have graded levels of satisfaction instead of being either satisfied or not. This allows to model more accurately competitive requirements.

Participatory Design (PD) is a user-centered technique (cf. the special issue on Participatory Design of the CACM (Kahn and Muller, 1993)), interested in an early user involvement in the development process. In contrast to requirements engineering, the origins of PD are aspects of social responsibility and studies of consequences of technology introduction to work places. The central idea is to involve the users in the development process to ensure that their requirements – which may be competitive to those of the management – are respected. Of course, ordinary users usually cannot specify nor implement systems, but they can, for instance, work with mock-up prototypes of user interfaces and assess them (Madsen and Aiken, 1993). The aforementioned viewpoint approach (Finkelstein et al., 1992) can be used as a framework to embed PD into a requirements engineering process. Contextual design (Holtzblatt and Beyer, 1993) is alternative approach, embedding users into the design process. Prototyping (Budde et al., 1992; Doberkat and Fox, 1989) is another well-known approach for early feedback of users in the development process which has some common roots with PD (Budde et al., 1992, p. 3).

The aforementioned approaches and models used in requirements engineering are in no way specifically tailored for multimedia applications. Some formal methods, however, are based on the same formal models as discussed in sec. 3.1.2.1 on page 29, e.g. the use of linear temporal logic by Păun and Chechik (1999). Also the use of prototypes and early user involvement deal with similar problems we discussed for multimedia applications. Methods and models explicitly devoted to hypermedia and multimedia are the topic of the next section.

3.3.2. Software Engineering based Hypermedia and Multimedia Design Methods

In the early 1990s some approaches for the methodical development of hypermedia or multimedia applications were published. The first group extends database approaches to hypermedia systems. The second deals with multimedia in a rather different way but interestingly in a similar field as our Altenberg Cathedral project.

3.3.2.1. Hypermedia Design Methods

Garzotto et al. (1993) present the hypertext design method HDM. Their approach is based on the idea of authoring-in-the-large reminiscent the programming-in-the-large concept of software engineering. HDM is an adoption of design approaches for database applications. Its core is a schema modeling all data used by the application. As the domain of hypertext requires, HDM focuses on different link types and on views called perspectives organizing the amount of data to be presented. Links are divided into perspective links, structural links and application link types. The former two link types connect either different perspectives of an information unit or structure components of an information unit, resulting e.g. in a hierarchy of components. The latter are defined by developers to meet the requirements of connecting different pieces of information depending on the content of the application domain. The link types easily allow for differentiation and relationship classification of information units. It is possible to derive even more links by applying graph operations such as closures. This can be done by an interpreter of the HDM schema.

The important contribution of HDM is to base the development of a hypertext application on a model, ensuring a more consistent system and resulting in a more structured and predictable development. As a hypertext method (and not hypermedia!), it is obvious that HDM does not take multimedia in account. Although it is (certainly) possible to incorporate still media such as images into HDM, all aspects of timed media and their specific needs for synchronization are not addressed.

RMM (Isakowitz et al., 1995), the Relationsip Management Methodology, is a hypermedia development model and method based on HDM. RMM uses an ER model similar to those of HDM including navigational aspects. RMM focuses on the methodology for developing applications. A four-step process is presented:

1. ER design: representing the application domain.
2. Slice design: defining information packages presented to and accessed by users.
3. Navigational design: designing paths through the information.
4. User-interface design: screen layout and appearance of objects from the datamodel.

Mentioned, but not discussed further, are requirements analysis, conversion of data-model elements into platform-dependent objects, runtime behavior design, construction and testing.

As the authors remark, RMM is well suited in application domains which are highly structured and deal with volatile data. In contrast, in areas such as the Altenberg Cathedral Project, which is more oriented towards humanities work and thus less structured, RMM is not useful as the authors argue (see also Lowe and Hall, 1999, p. 478). Interestingly, although RMM is designed for hypermedia applications, media aspects are only mentioned but not discussed (Isakowitz et al., 1995, introduction section): "multimedia aspects ... raise numerous difficulties". Isakowitz et al. (1995) do not address these aspects in the section about future work, so it seems as if in RMM any such difficulties are simply ignored.

OOHDM (Schwabe and Rossi, 1995; Schwabe et al., 1995), the Object Oriented Hypermedia Design Method, extends the concepts of HDM in various directions. Obviously, the schema is object-oriented based on OMT and is not merely an ER model. This is a minor change as HDM uses at least an extended ER model. More important is the developing process of OOHDM. Schwabe and Rossi present a four-step process:

1. Domain analysis: modeling the semantics of the application domain in a conceptual class model.

2. Navigation design: mapping between conceptual and navigational objects, taking user profiles and tasks into account.

3. Abstract interface design: modeling the user interface.

4. Implementation: constructing the executable application.

OOHDM emphasizes the whole initial development process (maintenance is not considered) where HDM focuses on the central data model. In particular, the abstract interface design allows the modeling of how different media object are used, and how the interaction with the user works. This is specified with abstract data views (Cowand and Lucena, 1995), which model user interfaces as compositions of primitive objects, such as buttons, and of media objects. The behavior is defined as the reaction to user and system events, and as the interaction between interface and navigational objects.

OOHDM is superior to HDM and RMM, with respect to the more expressive object-oriented data model and the support of different models for different aspects. For complex systems, this separation of concerns is important, because otherwise the resulting models are too complex for easy comprehension. But OOHDM also neglects problems of different media types. The abstract interface design allows the consideration of the media types and their representation more explicitly than the other approaches, but this involves only the aspects of the reaction to events. More general

considerations about media aspects are not available, especially as far as the temporal arrangement of media objects is concerned.

3.3.2.2. Communication and Multimedia Development

Morris and Finkelstein (1996, 1999) presented an unusual approach for software engineering support for multimedia applications (which is, by the way, independent of the viewpoint approach). The basic idea of their work is to study the use of media elements as communication vehicles from the author to the user in multimedia applications. In contrast to other approaches, the communication task of multimedia applications is explicitly considered and part of their model.

Morris and Finkelstein model a multimedia application as a combination of a document and a software system. Whereas the software always deals with physical media, during development the document itself also consists of abstract media, later to be coded in some physical media. These abstract media elements are parts of a sign system, theoretically described with formal linguistics and semiotics (analyzing documents with respect to their syntax, semantics and pragmatics, the latter dealing with structures larger than mere sentences). Thus, the content and structure of a document is analyzed on the basis of pragmatics. Typical structures in traditional documents are e.g. rhetorical figures such as the classic *thesis-antithesis-synthesis*. The discourse structure of a document is described in a similar way.

As traditional document types do not deal with different media types (e.g. books) or free positioning in the document by the user (e.g. movies), Morris and Finkelstein introduce a new type of discourse structures called *navigable discourse structure* (NDS). They define a NDS as an order relationship between abstract media elements together with a mechanism for the end-user to move between these media elements. This recalls classical hypertext models such as the Dexter model, but the focus of NDS is different. While hypertext models analyze existing documents with physical elements from a technical point of view, a NDS models the content of a document with its structure and media usage.

The NDS model is accompanied by a process model which is inspired by the cyclic nature of the well-known spiral model of Boehm (1988) but without applying risk analysis. The model consists of four stages or components:

1. Discourse structure (classical)
2. Abstract media elements
3. Presentation spreads and operations
4. Navigable discourse structure

In an iterated process, these artefacts are refined by transformation and reconstruction. The presentation spread is a large table relating characteristic media elements,

operations on these elements, and discourse components. It is similar to story boards (Harada et al., 1996). The operations are defined with a formal syntax but without intuitive or formal semantics (Morris and Finkelstein, 1996, sec. 8).

The main contribution of Morris and Finkelstein is their fresh view of document aspects during multimedia application development. They do not consider multimedia applications only as technically complex systems but also focus on the communication task, the content, of multimedia applications. The intended aid during production of multimedia applications is significantly hindered by the operations: their formal and sometimes cryptic syntax does not encourage their use by authors and the missing formal semantics hinders tool construction which could hide notation and their problems. The process mentioned is intended "to provide practical guidance for production", but remains quite abstract. No technical problems are addressed, additional work has to be done there. Related to our situation, this approach has the disadvantage that it ignores the temporal composition of media artifacts violating Requirement 3. Therefore, this approach is not suitable for our intention, although the focus on the discourse structure is very appealing in general.

3.3.2.3. Summary

The hypertext and hypermedia design methods discussed in the literature are well suited in domains which are highly structured and allow to build a domain model from which the other models are deduced. Because of their semi-formal notations they require technical know-how to deal with them.

The NDS approach is a direct counterpart to HDM and successors because NDS focuses on non-technical aspects by concentrating on the discourse structure of the multimedia document. It makes evident that there exists different kinds of multimedia applications which have very different requirements: document centric applications deal more explicitly with communication aspects and are less structured, whereas database centric applications are highly structured and restrain the communication aspects of each individual artifact by applying classification.

Interestingly both kinds of approaches neglect the multimedia aspect of synchronizing different media and are therefore not applicable in our context.

3.4. Assessment and Conclusion

In this section we asses the aforementioned approaches of the literature on the basis of our experience during the Altenberg Cathedral Project and the set of requirements we stated in sec. 1.1.4 on page 9. After that, we draw our conclusions in sec. 3.4.4 on page 47.

3.4.1. Formal Models

The formal models presented in sec. 3.1 address temporal behavior. All of them require sound mathematical knowledge and thereby adequately educated and trained users. However, formal models cannot be used as a common language for development, since they violate Requirement 1 (Understandability). They can, however, be the semantic basis for more user-oriented approaches. The systems from Goetze (1995) and Vazirgiannis (1999) are prominent examples of this. Both are in fact authoring environments and as such intended for implementing a multimedia system. In contrast to commercial authoring environments, they have a sound formal basis, specified with CSP and an algebraic approach, respectively. The underlying formal models are not visible to users. However, both system focus on implementing multimedia applications, and consequently they address technical oriented users and are not well suited for solving our problem of understandability among heterogeneous developer groups.

The formal document models have similar problems, when the requirement of understandability is addressed. Approaches such as the Dexter model are not intended for specification and address computer science researchers. HyTime and SMIL are well defined implementation languages, but our experience with ADML has shown that even XML based languages, i.e. languages with formal grammars, are not well suited for non-technical developers (Alfert et al., 1999).

From the logic-based approaches only Allen's interval logic is applied to multimedia. A reason for this might be that the temporal extension of media artefacts is immediately reflected by Allen's notion of intervals as primitive elements in his calculus. In contrast to this, in temporal or modal logics it is rather complicated to model such intervals (but not impossible as Hajnicz (1996) shows). Additionally, the relations between intervals in Allen's calculus are understandable even without a formal logical background. The constraint solving algorithm allows tool support, and extensions enable the incorporation of quantitative knowledge. Nevertheless, Allen's approach has some shortcomings. First, intervals are described on an instance level only, i.e. if some action is repeated twice, we have to describe this with two (independent) intervals. This makes it impossible to specify loops within Allen's calculus. Second, Allen does not deal with vague information, except for the duration of intervals, which he assumes to be unknown. Here, fuzzy temporal models are useful for modeling vague concepts, which we found in natural language settings and everyday reasoning.

Dubois et al. (2001, p. 391) suggested the use of fuzzy sets for giving preferences to the qualitative interval relations of Allen as a tool for temporal specification of multimedia applications. This is similar to our proposal (Alfert and Heiduck, 2002) developed independently.

3.4.2. Specification Techniques

Graphical formalisms (UML, Petri nets, statecharts) have gained some popularity because of their illustrative yet formal nature. This combination seems to satisfy two of the requirements mentioned earlier, common understandability (Req. 1) and (possible) tool support (Req. 4). While this is a clear advantage compared to, e.g., temporal logic, users require also a thorough training in these graphical languages.

Petri nets are currently often used as formal models of multimedia systems and seldom for specifying multimedia applications. For the latter, we have the familiar problem discussed also for the formal models: Petri nets require much training and despite their graphical nature they are not always as intuitive as one would wish. This especially conflicts with Requirement 1. This also holds for fuzzy timed Petri nets, since the only change with respect to more ordinary Petri nets is that the timing annotation is based on fuzzy set theory. Clearly, this helps modeling vague timing considerations, but the inherent complexity of Petri nets remains.

Statecharts and hypercharts have problems similar to Petri nets. The graphical representations give an intuitive feeling of understanding but nevertheless for their active use substantial training is required. In spite of the ability to decompose a model into a hierarchy of sub-models, it is hard to understand a complex model. Especially hypercharts, with their timing annotations of events, are not easy to read and may confuse the faint-hearted reader.

Similar problems arise with UML. While the formal underpinning of UML is not as strong as for Petri nets, its models for temporal behavior rely on statecharts and Petri net-like notations, and inherit their complexity. The OMMMA approach as an adaption and extension of UML is of no help here, since UML's temporal models comprehension is not improved, although the OMMMA's enhanced variants allow a better modeling of multimedia applications. This is not surprising, since OMMMA focuses on the traditional design phase of software development and thereby is interested in technical precision.

3.4.3. Software Engineering based Approaches

The approaches in requirements engineering and also in participative design are not specific for multimedia. They give, however, useful hints and insights. First, they support our opinion that including non-technical developers in the development process is required, useful and indeed feasible. Second, we should not aim at one single model, but allow for different models for different developers, as proposed by the viewpoint approach of Finkelstein et al. (1992), such that each developer can work with the best suited models. It is necessary, however, that the different models can be related to each other. Otherwise, (systematic) consistency checks between them are not possible and we have no real benefits compared to an uncoordinated development process.

The hypermedia development methods inspired by software engineering have major drawbacks in two areas. First, these methods neglect multimedia issues. Although supporting more media types than only text as in pure hypertext system, these methods particularly lack the means for specifying the temporal arrangement of media objects. Second, except for NDS, these methods require a conceptual domain data model. While this is a clear benefit for the development of all hypermedia applications based on information systems, it restricts the application of the methods in less structured domains. For instance, in the Altenberg Cathedral Project we do not have a domain model, hence the aforementioned methods are not applicable here.

3.4.4. Conclusion

Our review of approaches in the literature shows that no approach satisfies all four requirements. A particular problem is the requirement for understandability, which debases the formal methods and models. Additionally, this requirement seems to contradict the specification requirement, which most formal models satisfy. The requirement of understandability often triggers the introduction of vague concepts, which are poorly supported by most approaches, in particular those coming from software engineering.

To resolve the situation it is in our opinion useful to follow the advice of the viewpoints approach (Finkelstein et al., 1992) and to create different yet related models. In this situation we have found some useful “bits and pieces” in the literature, which, however, require some adaptation. In our approach, we divide the models in a dichotomy, one group with a sound formal basis and the other group without a sound formal basis. For the latter group we use natural language (NL), which is supported for the multimedia domain by Bailey et al. (2001b). But we have to be careful when dealing with the disadvantages of NL when formalizing NL specifications, as discussed earlier in sec. 1.3 on page 12. For the former group, two approaches provide first solutions:

- Allen’s interval logic offers tool support, an adequate model for the temporal extension of media objects, and is easily comprehensible. We require additional support for loops to use the calculus for specification purposes. Adding fuzzy set theory helps to deal with the imprecision introduced by natural language.
- Petri nets model synchronization aspects very well and can be used for specification. Tool support is also available. The imprecision of natural language is reflected by Mutara’s fuzzy timed Petri nets, which is another benefit.

The two formal approaches are related, since it is possible to model Allen’s calculus with Petri nets. This merits investigation of whether Murata’s fuzzy timing approach models back into the interval calculus. In the next chapter we develop our approach based on these considerations in more detail.

Part II.

Defining Vitruv

4. An Introduction to the Vitruv Approach

In this part, we define the languages of the Vitruv approach and develop their semantics. This chapter gives a brief introduction to the Vitruv approach and presents more details of the basic ideas already presented in sec. 1.2 and in sec. 1.3 on page 12.

In the first section (sec. 4.1), we recall the main ideas of the Vitruv approach. In sec. 4.2 on the next page we discuss some more details of Vitruv_{L}, after which we focus on the semantics of Vitruv_{L} (sec. 4.3 on page 53). The section is followed by a discussion of Vitruv_{N} in sec. 4.4 on page 55.

The sequence of sections in this chapter is the same as the respective sequence of chapters in the current part. We have chosen this order, since we prefer to provide the foundations before we move to Vitruv_{N}. For the same reason, we do not show any example of Vitruv_{L} or its semantics in this chapter, since any explanation of these examples would require an inadequate anticipation of many details, which are introduced later in this part. However, in sec. 4.4.2 on page 56 we present a simple example of a Vitruv_{N} specification, the formal counterparts of which are shown in the following chapters. We discuss a more complex example featuring Vitruv in its entirety in chapter 10, after presenting all technical details of Vitruv in the preceding chapters.

4.1. Basic Concepts of Vitruv

With Vitruv, we develop an approach for the specification of multimedia presentations, suitable for both technical and non-technical developers. In section 1.3, we presented a first glimpse on the structure of Vitruv, which we briefly recall here. To make the application of Vitruv feasible for non-technical developers, we use natural language (NL) as common base for communication. The subset of NL used is named Vitruv_{N}. To tackle the inherent problems of NL, namely ambiguity, imprecision, vagueness and incompleteness, we decided to provide a formal counterpart to Vitruv_{N}. However, giving a NL-based specification language a formal semantics is a difficult task and thus we decided to introduce a formal specification language named Vitruv_{L} as mediator between Vitruv_{N} and the formal semantics. Additionally, Vitruv_{L} serves as a specification language of its own for technical developers. The semantics of Vitruv_{L} are partitioned into three separate parts, 1) the static semantics, and the dynamic semantics for 2) event-free and 3) event-based behavior, respectively.

For the formalization of Vitruv_N we have several alternatives, as discussed in chapter 3, where we conclude that Petri nets and Allen's interval logic offer best support (sec. 3.4.4 on page 47). To ease the transition between these two languages we are interested in using similar abstractions in both. Allen's approach, originating from text understanding and everyday reasoning (cf. Allen and Hayes, 1985), is conceptually much closer to natural language than Petri nets, therefore we favor a formal specification language based on interval calculus. Hence, the key concepts of the formalization of Vitruv_N are temporal intervals and their relations (Allen, 1983) for representing synchronization issues of multimedia presentations. Additionally, we use fuzzy set theory for dealing formally with the inherent vagueness and imprecision of NL, needed in particular for durations of intervals.

4.2. Vitruv_L

Vitruv_L is the manifestation of the Vitruv_N formalization. We discuss its features followed by some remarks on conceptual modeling within Vitruv_L.

4.2.1. Language Features

Vitruv_L is a specification language of its own and thus has additional important features compared to Vitruv_N. We apply an object-oriented approach, such that all intervals are defined as instances of classes with logic clauses defining the (partial) order of intervals. Inheritance is interpreted as adding clauses, similar to the DoDL approach by Doberkat (1996) which in turn was inspired by the inner-construct of BETA (Madsen et al., 1993). As usual, classes may aggregate other classes, such that the composition of classes is based on both inheritance and aggregation. The class structure makes it possible to modularize and reuse Vitruv_L specifications.

With the interval calculus, we can only describe event-free behavior. For extending the behavior we introduce events, modeling user interaction. Reactions to events are considered by special rule sets, which control either loops or alternative branches. The latter are similar to case-statements in traditional programming languages. Loops and branchings are in turn regarded as intervals and thus can be ordered in the same way as ordinary actions. Event values are used for deciding whether loops are terminated or which branch is selected. The values are instances of fuzzy types. Fuzzy types are linguistic variables, defining a universe and a set of reference values and modifiers. In addition to event values, fuzzy types are used in particular for specifying the duration of intervals.

4.2.2. Conceptual Modeling and its Realization

The concept of linguistic variables allows the transformation of quantitative into qualitative data and th construction of conceptual models, since the specific realization (or valuation, as logicians would say) of such linguistic variables, i.e. their corresponding fuzzy sets, is of no immediate concern for the conceptual model. In fuzzy control theory (Yager and Filev, 1994), the realization of linguistic variables has to be adapted and fine-tuned for each new application, but leaves the conceptual model, consisting of linguistic variables, unchanged. We adopt this principle here. The conceptual models deal with qualitative data and are represented by linguistic variables. They remain stable during refinement of the specification by fine-tuning each fuzzy set, needed for developing the final application. This fine-tuning works on the realization (or valuation) level and binds each qualitative data to its realization as a fuzzy set. It also allows the binding of different realizations to the same conceptual model, which becomes a class of realizations in the same way as a type represents its possible values. This separation of conceptual models and their realizations implies that we gain more stable models since the conceptual models need no changes while applying different realizations.

The connection between the conceptual model and its realization is handled in the binding of $Vitruv_L$. In the binding we assign fuzzy sets and modifier functions to the reference values and modifiers of fuzzy types, respectively. Only in the binding, we instantiate classes, thereby creating objects. To realize aggregation, we use polymorphic assignments of objects.

4.3. The Semantics of $Vitruv_L$

The semantics of $Vitruv_L$ (and thereby also indirectly those of $Vitruv_N$) are split into three separate yet related areas. While moving from the static semantics to the two dynamic semantics (for event-free and event-based behavior), we add in each step a further amount of information and rely at the same time on the consistency provided by the semantics of the previous steps.

4.3.1. Static Semantics

The static semantics of $Vitruv_L$ defines formally the type structure and the scoping rules of $Vitruv_L$. We use a deduction system where the type and scoping rules are notated as the deduction system's inference rules. The application of the static semantics to a specific $Vitruv_L$ specification results in finding a formal proof deriving that the specification is well-typed. The type system of $Vitruv_L$ provides static and strong typing, i.e. each entity is required to have a type and all type errors can be found by

analyzing the specification. As usual for the object-oriented setting, we allow inheritance induced polymorphism for variables with class types.

The benefit of a static and strong type system for Vitruv_{L} is that we can prevent many errors in advance in the specification by type checking. In particular, this is helpful for large specifications, and for specifications consisting of separated and independent parts. The latter occurs frequently in Vitruv_{L}, because classes are used for modularization of Vitruv_{L} specifications. Additionally, the distinction between the conceptual specification and the binding, which is also separated in the specification text, is a potential source of errors, which can be detected easily by type checking. From a pragmatic point of view, the binding in its current form is only feasible because of the advantage of having a static and strong type system.

4.3.2. Dynamic Semantics for Event-Free Behavior

The first dynamic semantics is concerned with the event-free behavior specified in Vitruv_{L}. The event-free behavior is the part of the dynamic behavior of Vitruv_{L}, but independent of events, selectors and loops. The interesting fact of the event-free behavior is that we can check its consistency with a variety of algorithms, beginning with Allen's original constraint solver. The consistency checkers and constraint solvers give us the ability to infer unspecified or refine specified durations and interval relations, and to check for contradictions in the specification.

However, these algorithms require a flat set of intervals, relations and duration constraints. Therefore, it is not possible to use directly the definitions of Vitruv_{L}, but we need a linearized version of them. To ease the process of providing a linearized version of a Vitruv_{L} specification, we define an intermediate language called Vitruv_{I}, which handles only intervals, durations and interval relationships, i.e. the vocabulary for the event-free behavior. The linearization process depends on the statics semantics, since we need not only a well-typed specification but also information derived during type checking. The linearization of a Vitruv_{L} specification results in a Vitruv_{I} program, which has to be executed as final step of the linearization. After execution, the flat set of intervals, their duration constraints and interval relations are provided. The execution of the Vitruv_{I} program is formally defined by an operational semantics. The semantic domain used for the state of the operational semantics contains the set of intervals, their duration constraints and the interval relations of the Vitruv_{I} program, and can be used as input for consistency checking algorithms.

4.3.3. Dynamic Semantics for Event-Based Behavior

The second dynamic semantics handles not only the event-free but also the event-based behavior specified in Vitruv_{L}. Hence, we support events, loops and selectors, not available in the semantics for the event-free behavior.

We define the dynamic semantics by a variant of timed colored Petri nets which we call *Vitruvian Nets*. In Vitruvian Nets we combine several Petri net types, all of which are well-suited for particular aspects of the behavior of Vitruv$_L$. Timed Petri nets are suitable for modeling Allen's interval relations (Little and Ghafoor, 1990), and with Fuzzy Timed Petri Nets (Murata, 1996) we support fuzzy durations of intervals. Events, which model user input in Vitruv$_L$, are realized by transitions, which fire tokens with random values. The tokens are fed into decision processes used in selectors and loops. These processes are modeled by Fuzzy Petri nets, where tokens take fuzzy values (Looney, 1988). The repetition and branching nature of loops and selectors, resp., are supported by the net topology of Petri nets in general. Constructing the net topology requires the linearization developed for the semantics of the event-free behavior. This is the connection between the two dynamic semantics and thereby indirectly also the connection to the static semantics.

Defining the semantics of the event-based behavior of Vitruv$_L$ with Petri nets allows the use of analysis techniques for Petri nets to analyze the Vitruv$_L$ specification, e.g. for determining properties such as liveness, deadlock situations or reachability. By simulating the net we can analyze additional properties, which depend on the random firing of transitions modeling events, e.g. average durations of intervals or paths. The simulation capability can also be used for tools providing an early and user-friendly feedback to technical and non-technical developers: it is important that the formal notation of the Petri net is translated back to abstractions which are familiar to the developers. The details are left to an implementation.

4.4. Vitruv$_N$

The part of Vitruv for specifying multimedia presentations usable by technical and non-technical developers is Vitruv$_N$, a specification language based on natural language (NL). We discuss the features of Vitruv$_N$ and give an example.

4.4.1. Language Features

The structure of Vitruv$_N$ follows those of storyboards, such that in each Vitruv$_N$ document an entire presentation is specified. Each presentation features scenes as the highest structural element. Inside each scene, a set of media objects is used. Media objects are required to have a certain media type, e.g. video, audio, image etc. The main part of a scene description specifies the temporal arrangement of the scene's media objects. Some media objects may have an internal temporal structure, e.g. various shots inside a video clip. In such cases, we can use also use the internal media structure in the same way for the definition of the temporal arrangement.

The specification of the temporal arrangement of media objects is given primarily by stating how media objects are mutually related. This is quite similar to Allen's

interval calculus, making the translation to Vitruv_L easy, yet providing an adequate and not too artificial description in NL. Compared to Vitruv_L, the features in Vitruv_N for handling events are somewhat simpler. We decided on this option, since we did not want to bother non-technical developers with the many technical details needed for more complex control structures.

For Vitruv_N, we decided to use a rather restricted subset of NL, restricting both vocabulary and grammar. As a result, we gain a (general) context-free grammar for Vitruv_N. The restriction has its benefits, reducing the risk of ambiguity inherent in NL and easing the construction of tools for Vitruv_N. As a disadvantage one may argue that a restricted vocabulary and grammar makes it difficult and error-prone to write specifications. But since specification documents are much often read than written, we favor an approach focusing on readability. To ease writing documents, however, with the particular vocabulary and accompanying grammar rules of Vitruv_N, we can support the editing process with user-friendly tools, such as syntax-oriented editors. The context-free grammar used allows to build parsers as the first step towards machine-readability and general tool support.

The semantics of Vitruv_N are given implicitly by the semantics of Vitruv_L. Thus, we are required to provide a mapping from Vitruv_N to Vitruv_L. Due to the regular structure of Vitruv_N, its restricted grammar and vocabulary, and its proximity to behavioral aspects of Vitruv_L, we can define a systematic translation from Vitruv_N to Vitruv_L.

4.4.2. Example

In the following example 4.1 on the facing page we present a simple scene, consisting of three media objects and their relations. As a reading hint, we should note that identifiers in Vitruv_N are delimited by quotes to allow several words long identifers, in the text we additionally use angle brackets as delimiters to distinguish Vitruv_N code from Vitruv_L or Vitruv_I code. The scene starts with rendering of video ⟨“house”⟩ and finishes with video ⟨“house2”⟩. Both videos are overlapped by the audio-clip ⟨“transition”⟩, which shall accompany the move from the first video to the second.

The example shows only a small fraction of the possibilities of Vitruv_N. But this is the reason why we use it also as introductory example for Vitruv_L (spec. 5.2 on page 65) and for Vitruvian Nets (fig. 8.4 on page 162) in the respective chapters.

4.5. Sketching a Process Model for Vitruv

For applying Vitruv we propose a process model sketch, which is strongly related to the distinction between conceptual model and its realization, as it is made explicit in Vitruv_L. The process model consists of the following steps:

1. Specification of the presentation with Vitruv_N

Specification 4.1 A simple example scene

Description of scene "Example".

In this scene, these media are used:

- A video, identified by "house1".
- A video, in the following named "house2".
- An audio-clip identified by "transition".

End of media definition.

The media composition:

The scene starts with video "house1". Video "house1" and audio-clip "transition" overlap slightly. Video "house2" and audio-clip "transition" overlap slightly. The scene finishes with video "house2".

End of the media composition.

End of the scene description.

2. Translation to Vitruv_L
3. Consistency checking and net generation
4. Net simulation
5. If the behavior of the net is not satisfactory: fitting of the binding and go to step 3
6. Finish.

It should be remarked that this sequence of steps is an ideal similar to the waterfall model. In practice, backward links to all steps might become necessary. Additionally, tool support is appreciated.

The first step establishes the initial specification in Vitruv_N, understandable for all developers, technical and non-technical. Thus, this specification is the communication vehicle for the entire development process. In the next step this specification is translated to Vitruv_L. The Vitruv_L specification is not intended to be used by non-technical developers, we focus here on technical developers only.

Steps 3 to 5 are repeated until all developers are satisfied with the specification. In step 3, we check the consistency of the Vitruv_L specification and transform the specification to a Vitruvian Net. As next step, we simulate this net. The purpose of the simulation is to check whether the behavior of the system meets the expectations of all developers. If this is not the case, then the specification has to be changed. Our modeling approach assumes that the qualitative part of the specification remains stable and

only the quantitative part requires changes, demanding probably a new requirements elicitation. Technically, it should be sufficient to modify bindings for media durations, but alternatively we can modify media durations in the Vitruv_N specification as well. After modifying the durations, we go back to step 3 (or to step 2, if we modified the Vitruv_N specification), and repeat the remaining steps.

We finish with step 6 after the simulation results are satisfactory. We consider now the specification as stable. The design and implementation of the presentation can begin.

5. The specification language Vitruv_L

5.1. Overview

In Vitruv_L we want to capture central aspects of Vitruv in formal way. In particular, we are interested in

- the ordering of activities, i.e., temporal relationships between activities,
- the duration of activities,
- events, their temporal interval in which they may occur, and their relationships to other activities and events, and finally
- reactions to events.

Activities are time-consuming actions. Intuitively, we can model them in a very abstract way as closed temporal intervals. The relationships evolve naturally in interval relationships as presented by Allen (1983) (sec. 5.2.1 on the next page).

However, Allen's calculus is only a set of formulas constraining a set of intervals. It is a flat system requiring a structuring mechanism, if we aim at maintainability and feasibility of large specifications. In sec. 5.3 on page 62 we apply object-oriented concepts for structuring the specification.

Since the Vitruv approach is based on natural language, we have to deal with imprecise descriptions such as rather vague measures of durations, e.g. short or long. As discussed earlier, requirements, stating that certain actions have a long duration with a common-sense semantics – needed for the natural language basis of Vitruv– , do not mean that these actions have exactly the same duration but durations which are more or less compatible to a distinguished reference value. Hence, the compatibility does not define an equality or a classical equivalence relation rather consists of graded truth values stating whether a specific duration fits to the reference value. Fuzzy set theory is the well-established theoretical underpinning of such vague concepts. Applied to our interval-based approach we introduce fuzzy intervals, which have fuzzy durations, and fuzzy relationships extending Allen's calculus. In sec. 5.4 on page 67 we present user-defined fuzzy types such as durations.

With Allen's calculus we can only have a pre-determined temporal behavior without any user interaction. However, in interactive multimedia presentations users control the behavior of by following links, choosing between menu items, starting or

stopping of media renderings such as videos, and so on. These interaction can be modeled by events and reactions to events. In sec. 5.5 on page 70 we present events in $Vitruv_L$. The system's reaction to events are considered by special rule-sets allowing if-then-like clauses and loops. Additionally, user interactions such as following links or using menus require the possibility for branching to different parts of the presentation. This is discussed in sec. 5.6 on page 78, where we concentrate on the presentation's structure in the large. A library of standard definitions is presented in sec. 5.7 on page 79.

In fuzzy control theory (Yager and Filev, 1994) it is important to distinguish between the conceptual expressions (such as long or short) and their values (the respective fuzzy sets). One of the appealing characteristics of fuzzy control theory is that usually only the values need a fitting for a particular application whereas the conceptual expressions remain constant. In order to transfer this characteristic to $Vitruv_L$, we separate classes, relations and other constraints from their realizations (sec. 5.8 on page 81). In particular, this is useful because fuzzy durations such as *short* or *long* are context-dependent and hence their values might differ considerably.

Finally, we should mention that we discuss some possible extensions of $Vitruv_L$ in sec. 12.2 on page 256. They are not part of the language definition because they do not address the language core, they provide however valuable concepts for a more pragmatic language use.

5.2. Intervals and their Relationships

The most basic materials in $Vitruv_L$ are intervals and their relationships. They are discussed in this section.

5.2.1. Intervals

intervals

alpha

omega

length

Temporal *intervals* are the base abstractions of $Vitruv_L$ and are based on the real line as time scale. Each interval is characterized by its definitive starting and finishing point, *alpha* and *omega*, resp. The distance between these two points is the interval's duration, its *length*. These three characteristics are attributes of each interval in $Vitruv_L$ and can be accessed by the usual dot-notation: let x be an interval, then the length of x is notated as x.length, its starting and finishing point as x.alpha and x.omega, resp. These attributes are an augmentation of Allen's approach which considers intervals only qualitatively. But we want to quantify intervals and later also the interval relationships, thus we need these attributes.

activated interval

Let us now relate intervals and activities. Each interval i in $Vitruv_L$ corresponds to an activity a. Whenever the activity a is executed, we say its corresponding interval i is an *activated interval* and *vice versa*. Similarly, the behavioral characteristic of

an interval is related to the behavior of its corresponding activity. For intervals representing (time-dependent) media objects the corresponding activity is the rendering of the media object and the intervals' length is accordingly the rendering time. The order of activities (and hence the order of intervals) is defined by interval relationships, explained next.

5.2.2. Interval Relationships

Interval relationships specify the partial order of intervals, thereby controlling the activation of intervals and thus (an important part of) the behavior of the presentation.

Interval relationships are either primitive or modified relations. A *primitive relation* is fuzzy version of one of Allen's thirteen interval relations (cf. fig. 3.1 on page 31), i.e., they are binary fuzzy relations of interval positions. Modified relations are based on a primitive relation to which one or more modifiers are applied. We call such a relation a *compound relation*.

primitive relation

compound relation

The intended semantics of modifiers in compound relations requires the introduction of additional anonymous intervals, additional constraints and relations between these new intervals and those being arguments of the compound relation. The additional intervals are anonymous, because they are not visible and accessible outside the compound relation. As an example consider A shortly after B which introduces a new interval C met by A and meeting B together with a length constraint C is short.

Thus, we consider compound relations as a kind of macro which expands to a set of constraints including primitive interval relations. The set of possible compound relations is not fixed but can extended by the user. This implies that we have to define compound relations explicitly. Specification 5.1 shows the definition of shortly after. The intervals A and B are formal parameters, C is a new anonymous interval. We omit any typing of A, B and C, because local variables and formal parameters in a compound relation have always type Interval. C gets the length constraint that C is short. We defer details about the length constraint short to sec. 5.4 on page 67, where we discuss fuzzy types and values.

Specification 5.1 Definition of the compound relation shortly after.

```
define interval relation A shortly after B =
let C in rules
  B meets C;
  C meets A;
  C is short;
end;
```

5.3. Classes Structuring the Specification

Until now we have presented rather unstructured specifications. Intervals as a kind of runtime concepts are not well suited as structuring mechanisms for specifications. Nevertheless, intervals themselves have structure and also enjoy relationships to other intervals, which reminds of objects in object-oriented languages: objects are runtime concepts, have structure and relationships. Because of that we now apply object-oriented concepts: classes and inheritance for describing the structure of intervals, objects as runtime concepts represent intervals.

5.3.1. Classes

We are interested in an explicit modeling device structuring the otherwise flat set of intervals and their relationships. This device should fulfill the following general requirements regarding modularity:

- information hiding
- compositionality
- explicit interfaces

These features are main properties of abstract data types (ADT). One specific realization of ADTs are object-oriented classes which have been proven to satisfy these requirements. However, since it is well-known that inheritance breaks encapsulation it is required to balance object-oriented and ADT features. Nevertheless, the conceptual proximity of classes to typical implementation languages in the multimedia domain is another benefit. Therefore, we will use here object-oriented concepts. Firstly, we discuss important general properties of classes, followed by details of syntax and some selected concepts.

class
instance

In $Vitruv_L$, a *class* defines a set of objects with identical structure and internal behavior. We say that an object is an *instance* of a class. Each object represents semantically an interval and may have a complex internal structure. A class consists of two parts. The first part structures the class by defining elements of which the class consists. These *elements* are either objects or instances of other types, i.e. of fuzzy types (see sec. 5.4 on page 67) or event types (see sec. 5.5.1 on page 70). This part shows composition of classes from other classes, extended by inheritance later. The class's second part defines the internal behavior. It consists of a set of *rules* subsuming interval relationships, duration constraints and event reactions applied to the elements of the class.

elements

rules

Syntactically, these two parts are divided into four sections:

local declarations
let

1. *local declarations* defining the elements constituting a class. They are introduced by the keyword let.

2. an *export clause* declaring elements visible to the outside the class. The clause starts with keyword exports.

export clause
exports

3. an *inheritance declaration* – we postpone its discussion to section 5.3.2 on page 65.

4. a *body* defining the rules of the class and thus the behavior of the class. The body is enclosed by keywords body and end.

body
body

In contrast to general imperative or functional object-oriented languages, we do not provide methods, any behavior is defined within the set of rules. Because each object represents a temporal interval, the aggregation structure of objects is a temporal hierarchical structure of objects, implying that an interval, representing an outer object, contains all intervals representing the inner objects. The (static) activation of an object is consequently determined by interval relationships as explained on page 61. In addition to that, events, loops and selectors deal with dynamic and event-based activation of objects. Hence, there is no need for defining methods explicitly, and a finer grain of control over the behavior can be achieved by accessing (and thus activating) elements of a class by referring to them in the rules. As the body of class has a procedural character, classes in Vitruv_L are more similar to patterns of BETA (Madsen et al., 1993, p. 42) than traditional classes, because patterns as unification of classes, procedures and variables can also have a procedural character.

Whether access to elements from outside the class is allowed, depends on the accessibility status of each element. Generally, any access to elements of classes is forbidden for other classes, except to those which are explicitly exported by their declaring class. All exported elements of a class define the *interface* of this class. If the export clause is omitted in a class, only inherited exported attributes are exported (see sec. 5.3.2 on page 65 for the discussion of inheritance in Vitruv_L). Additionally, no rule is accessible in any way outside the class, hiding information about internals of classes.

interface

If elements of a class, declared in the local declarations, are instances of classes, then we call them *attributes*. To achieve safer specifications in Vitruv_L, we apply typing and declare explicitly the class for each attributes. If no new elements are declared in the local declarations, then the rules in the class body can only operate on inherited elements.

attributes

As usual in object-oriented languages it is possible to define recursive classes, i.e. the class can be used as type of its own elements. But in contrast to usual object-oriented approaches, we have only value semantics and no references (with the exception of references to scenes, cf. sec. 5.6 on page 78). Thereby, in contrast to recursive classes, we do not have recursive objects. The finiteness of the binding ensures that we do not get infinite objects, independent of any recursive class structure. This is similar to the definition of recursive data structures in SML without using references (Reade, 1989, p. 155 and following). The special attribute this is used to denote the

this

current object of a class, similar to this in Java (Gosling and Arnold, 1996) or current in Eiffel (Meyer, 1992).

The body of a class declares the rules, i.e. all relationships between elements, duration constraints and event reactions. All rules in the body of a class are implicitly connected by conjunctions. The semicolon at the end of each rule can be regarded as a logical-and with lowest priority of all operands: a set of rules R1; R2; R3; is then semantically equivalent to (R1) and (R2) and (R3). Each element e of a class C is only active while the actual object of class C is active. We require implicitly that between e and the actual object this of class C the relationship

e (during or starts or finishes) this

holds. This implies that the duration of each element is not greater than the duration of this.

The rules in the body of classes make use of the recursive structure of classes, which leads to the question, how these rules are applied to objects which are finite in contrast to the possible infiniteness induced by the class definition. We denote objects, that are not instantiated but mentioned in rules, as ***nil objects***. The evaluation of rules with nil objects depends on the kind of rule:

nil objects

1. Interval relations defining the arrangement of two intervals make only sense if both intervals exist; arrangements with non-existing objects are meaningless. Therefore, any interval relations where at least one nil object is used, are ignored.
2. In constraints, where the target of the constraint (the left-hand-side) does not exist, are ignored, because constraints for non-existing objects are meaningless.
3. In constraints, where the constraint expression (the right-hand-side) contains nil objects, the constraint value cannot be calculated, the value is undetermined and thus unknown. In our possibility theory setting, such a value means that all values are possible, because we cannot establish any restrictions. This is expressed by the constant 1-fuzzy set $\mu(x) = 1 \; \forall x$, stating formally that each x is equally and fully possible.

Specification 5.2 on the next page presents a simple example class specifying that two videos are rendered sequentially with an audio transition between both videos. We use the predefined attributes alpha and omega for beginning and end, respectively. The three attributes house1, house2 and transition are instances of the classes video and sound, resp., defined elsewhere. Only attributes house1 and house2 are exported and therefore accessible outside class Example. We use again the usual dot-notation to access elements of a class. The body defines the behavior: the sequence of the videos house1 and house2 with a sound transition guiding from the first video to the second. The sound is rendered parallel to the end of video house1 and to the beginning of video house2.

Specification 5.2 A simple class.

```
class Example
  exports house1 , house2;
  let
    house1 : video;
    house2 : video;
    transition : sound;
  body
    alpha starts house1;
    house1 overlaps slightly transition;
    transition overlaps slightly house2;
    house2 meets omega;
  end;
end;
```

5.3.2. Inheritance

Inheritance is the object-oriented concept for building and using a classification hierarchy of classes. In contrast to the division of classes into smaller parts that we have seen in the last section, inheritance works in the opposite direction, synthesizing new classes by inheriting features from other classes and adding new features. The introduction of inheritance in Vitruv_L demands to deal with the topics polymorphism, type rules, visibility of class elements, and late binding. We discuss these topics now.

5.3.2.1. Defining Inheritance

Following Cardelli and Wegner (1985), inheritance defines a subset-relationship between sub- and super-classes, regarding classes as sets of values and thus as types. Because each sub-class instance is also an element of the super-class, the subset-relationship suggests that instances of a sub-class can be applied in all situations where an instance of the super-class is expected. This is called substitutability and is the basis of polymorphism. To ensure this, certain conditions have to be established. In our situation, we do not have functions with complex type-rules, therefore the situation is much easier. We have only to ensure that (1) interfaces of sub-classes are an extension of their super-classes' interfaces and that (2) sub-classes do not weaken inherited rules. The first condition applies to the sub-typing rules by Cardelli and Wegner (1985), because the set of exported elements of a class defines essentially the type of this class. The second condition refers to the design-by-contract rules of Meyer (1997) for logical specification of classes' semantics, in particular those for class invariants.

Therefore, we require that the following two conditions hold:

1. all exported elements of a class are also exported unchanged by its sub-classes, guaranteeing that the interface of a class is always completely part of the interface of its sub-classes.
2. the set of rules defined in the body of a sub-class are combined by a logical conjunction with the rules of its respective super-classes. Therefore, a sub-class has to fulfill all laws defined by its ancestors.

These two conditions ensure that substitutability is always possible in $Vitruv_L$ and because of that inheritance is properly defined according to Cardelli and Wegner (1985).

Let us now explain inheritance in $Vitruv_L$ in greater detail. The effect of inheritance is that the new class contains all structures (i.e., declarations and rules) defined in the inherited class. Additionally, the new class can define its own set of local declarations and rules between new and old elements. The new elements need fresh names to avoid naming conflicts. The inherited exported definitions are also always exported by the new class. We allow only single inheritance avoiding harmful conflicts such as name clashes by combining independent hierarchies. We additionally gain a simple and strict specialization hierarchy between classes instead of a directed acyclic graph.

5.3.2.2. Class Interval: Root of the Inheritance Hierarchy

Interval

All classes inherit at least implicitly or indirectly from Interval, the root of the specialization hierarchy. Obviously, Interval does not inherit itself, as hierarchies are not reflexive! The definition of the class Interval (see spec. 5.3 on the facing page) serves similar aims as the class Object of Java (Gosling and Arnold, 1996): stating some general features which are present in every other class. Additionally, class Interval ensures that intervals are indeed the base abstraction of $Vitruv_L$ and that intervals are realized by objects. The specification of class Interval is straightforward as it only states that it consists of alpha and omega, and provides the special attribute length, which is a fuzzy variable of the standard fuzzy type DURATION (sec. 5.4.2 on page 69), maintaining a constraint on the fuzzy difference of alpha and omega.

However, both attributes alpha and omega are special, because they are considered atomic, i.e. they do not contain any other intervals. This is indicated in the specification of class Interval by stating that alpha and omega have length zero, but we explicitly forbid any other interval occurring in alpha and omega.

5.3.2.3. Late Binding

Semantically, inheritance works in $Vitruv_L$ as a macro expansion mechanism because we can simply collect for each class all structures defined by its super-classes throughout the entire hierarchy. This results in flattened classes consisting of all structures induced by inheritance. The definitions of these flattened classes are independent with respect to the inheritance relationship. Other relationships to other classes may

Specification 5.3 The definition of the class Interval.

```
class Interval
  exports alpha, omega, length;
  let
    alpha  : Interval;
    omega  : Interval;
    length : DURATION;
  body
    alpha starts this;
    omega finishes this;
    alpha (meets or before) omega;
    alpha.length is DURATION.zero;
    omega.length is DURATION.zero;
  end;
end
```

exist only if a class declares a local element of another class. Assignments to such elements are possible in a polymorphic way because of substitutability, this is done in the binding which is discussed later (see sec. 5.8 on page 81). This leads to a last point, which we need to discuss: *late binding*. Traditionally, it deals with determining the right implementation of a method of an object at runtime depending on polymorphic assignments of an object to an variable. In Vitruv_L we do not have functions or methods as mentioned earlier, but the rules serve a similar purpose, because they refer to elements. The temporal order of interval activations inside an element of a class depends on the element's actual value. That means, we may rely on the exported elements of such a class, but we can not rely on the temporal order of activation of such exported elements, because this depends on the actual polymorphic assignment of this element.

5.4. Fuzzy Types

In the section concerning interval relationships we used linguistic variables implicitly in the rules. In Vitruv_L such linguistic variables are called *fuzzy types*. In this section we describe their structure and properties, and present two standard types.

fuzzy types

5.4.1. Structure and Properties of Fuzzy Types

In Vitruv_L we use fuzzy types together with classes for the static structure. Similar to types in usual programming or specification languages, a fuzzy type defines a set of values. A value of a fuzzy type is a fuzzy set on a *universe* which is the basis of the

universe

type. A reason to introduce fuzzy types in Vitruv_L is to have a possibility to define durations within Vitruv_L and to have a means for data definition different from defining intervals. However, the simple structure of our fuzzy types limits the expressiveness of the data definition. Additionally, for the sake of simplicity, we restrict ourselves to simple universes, because they are sufficient for our focus of temporal specifications. Thus, the universes are either (closed) intervals of $\mathbb{R}$ or $\mathbb{Z}$, or a set of enumerated identifiers. The last alternative is useful for choices in menus of user interfaces. It may be irritating that we use fuzzy types for such discrete values, but singleton sets with a non-empty core encode discrete values. Thus, we do not need an additional non-fuzzy construction. In sec. 12.2.1 on page 256 we discuss more elaborated constructions for universes of fuzzy types.

terms
modifier

In addition to the universe, a fuzzy type consists of a set of reference values (i.e. fuzzy sets on the universe) and a set of modifiers. The reference values are called ***terms*** in Vitruv_L and are the literals of the fuzzy type. A ***modifier*** is a unary function which takes a fuzzy set as an argument and returns a new fuzzy set. Since a suitable definition of modifiers is generally situation and application domain dependent, it is important that each fuzzy type defines the set of applicable modifiers to its values. The negation of a fuzzy set is also a modifier, because it is an unary function.

Expressions of a certain fuzzy type consists of terms, modifiers applied to an expression, and the union and intersection of expressions. The latter are expressed by union and intersect, resp., and also by or and and, because these operator names are conventionally used in the literature. These set theoretic operations can be applied because each value of fuzzy type is always a fuzzy set as mentioned above. Expressions can be used for at least two purposes: firstly, they denote a fuzzy, an imprecise value and secondly, they constrain a variable of the same universe by denoting the (graded) set of possible values.

As an example for applying fuzzy types for something other than temporal constraints – these are shown in the next section –, we define a fuzzy type Brightness in specification 5.4 on the facing page. This type may be used to describe the brightness of visual elements. Its universe is the integer interval $[0, 255]$ in accordance to the usual rgb color systems in video displays. The terms, the reference values, are black, dark, muddy, shining, bright and white, the allowed modifiers are very, more_or_less and not. Note that we only define names and omit any assignments of (explicit) fuzzy sets or function definitions to these names. The assignment of such values is done in the binding (sec. 5.8.6 on page 84), where the realization of the fuzzy type is defined. The only exception is the universe definition, which is given explicitly and fixed, whereas all other values referred to in a type definition are left to the interpretation in the binding.

Specification 5.4 The fuzzy type Brightness.

```
define type Brightness
  universe [0 .. 255];
  define term black , dark , muddy, shining , bright , white;
  define modifier very , more_or_less , not;
end;
```

5.4.2. The Standard Type DURATION

In specification 5.1 on page 61 we used the term short as a value assigned to the length of interval C. Term short was only implicitly defined, but in spec. 5.3 on page 67 we defined that the length of an interval has type DURATION, hence short has to be a term of this type. Now we define the standard fuzzy type DURATION, which is used for all calculations and constraints concerning the duration of intervals (spec. 5.5). DURATION's universe is the subset of non-negative real numbers ($\mathbb{R}_0^+$), each value is a fuzzy number or interval, i.e., a normalized convex fuzzy set (def. B.21 on page 286). We provide the terms zero, short and long as generic durations. The modifiers allowed for durations are not, very, extremely, more_or_less and slightly.

DURATION

Union and intersection of fuzzy sets do not guarantee convexity (for intersection only if one of the arguments is not convex). Therefore, we have to be careful if we apply these standard operations to values of the fuzzy type DURATION, because we may achieve results which are not proper values of type DURATION.

In general, terms such as short have a context dependent meaning: a part of a movie may be seen as short compared to the full-length movie, but may be large compared to a single scene of the movie. In a programming language setting, a classical solution for the context sensitivity of fuzzy type terms would be to realize terms as functions, where the argument is the fuzzy set defining the context. In Vitruv_L we do not have this possibility, however, we introduce a notion of context in the binding (see sec. 5.8.4 on page 83), such that a term may have different realizations depending on the current context. A standard binding is provided in the prelude (sec. A.3 on page 273), where as context an entire scene is assumed.

Specification 5.5 The fuzzy type DURATION.

```
define type DURATION
   universe [0.0 .. inf] ;
   define term zero , short , long;
   define modifier not , very , extremely , more_or_less , slightly;
end;
```

5.5. Events, Loops and Branching

Till now, we can only specify pre-orchestrated presentations, i.e. its behavior is completely fixed and no user interaction can occur, at least not altering the behavior in any way. This is certainly not realistic, and thus we now introduce events and reaction to events, resulting in loops and branchings.

5.5.1. Events

events

In Vitruv_L *events* model user interaction. Each event consists of three parts:

- an enabling interval, in which the event may occur,
- an occurrence interval, in which the event occurs and which is a subinterval of the enabling interval, and
- a value of a given (fuzzy) type.

Events have to be declared as elements of a class, because their enabling interval can be positioned according to other elements in the class, requiring a suitable type for events. The structure of events is always the same. The only possible difference is the value's type, because enabling and occurrence intervals are simple intervals. Hence, events are generic type definitions such as arrays in Pascal or generic classes in Eiffel (Meyer, 1997, chapter 10). We follow the convention of these languages and use square brackets for denoting the value's type. This is shown in spec. 5.6 on the facing page, where we specify fuzzy type ButtonState and class Button. The latter declares the event pressed with value type ButtonState.

The enabling interval of an event usually has a length greater than 0 and is determined by the relation to other intervals. In contrast, the length of the occurrence interval is always 0, because events are characterized as time-points without temporal extension. The event occurs during the enabling interval including start and end points of enabling interval. Thus, between occurrence and enabling interval the relation starts or finishes or during holds implicitly.

Occurrence time and values of events are nondeterministic by nature. Hence, any relationships between ordinary intervals and occurrence intervals would transfer this non-determinism to the activation of these ordinary intervals. While this seems be a possibility to model if-then-clauses, it is very problematic. Consider the following situation:

$$a \text{ finishedby } b;\ e \text{ meets } b;\ b \text{ meets } c;\ a \text{ meets } c;$$

where e is an event's occurrence interval and a, b and c are ordinary intervals. b can only occur if e occurs, and this is propagated to c. This seems to be no problem, because b and c are immediately and explicitly connected by the meets-relation. But this connection exists for other intervals as well because we can infer further relations

Specification 5.6 Declaration of event pressed in class Button.

```
define type ButtonState
  universe { up, down };
  define term isPressed;
end;

class Button
  exports pressed, ...;
  let
    pressed : Event [ButtonState];
    ...
  body
    this.alpha start pressed; // only the enabling interval!
    ...
  end;
end;
```

with Allen's algorithm. This situation is shown in fig. 5.1 on the next page, including the following extension of our example by

$$d \text{ after } a;$$

implying

$$d \text{ after } b;$$

However, this means that there exists an interval i with

$$b \text{ meets } i; \; i \text{ meets } d;$$

and this makes d again dependent on e, which neither seems to match any intuitive semantics nor seems to be useful at all, because with the existence of e any intervals occurring after it would also depend on e. Additionally, the dependency even works backward. Any intervals in relationship with e can occur only if e occurs. In our example a depends via b on e, requiring that a can occur only, if e happens. But this breaks causality, because now the future determines the past. This must not happen.

Therefore, we have to restrict the relations permitted to events. All ordinary intervals may only refer to the enabling interval of an event. The enabling interval is not automatically finished if the occurrence interval is activated, thereby the enabling interval is an interval with a deterministic length and thus behaves like ordinary intervals. The occurrence interval and the value depend on each other. If the event occurs, then the occurrence interval is activated and the event's value is set. If the user interaction does not happen, we define that then the occurrence interval finishes the enabling interval. For this case, we introduce the distinguishable TimeOut-value as

TimeOut

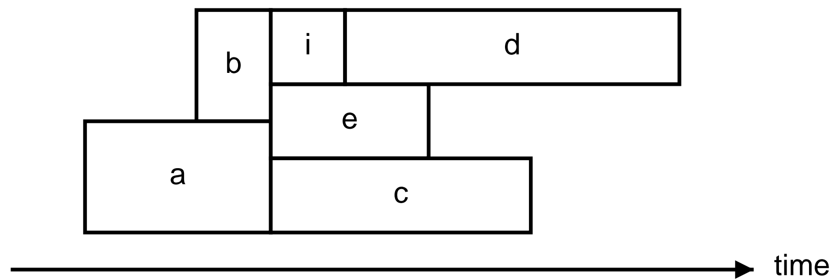

Figure 5.1.: The temporal arrangement of a, b, c, d, e and i.

time-out event

value

the event's value, which is not allowed to appear as an ordinary value in all fuzzy types. We call such an event a ***time-out event***. It ensures that the occurrence interval always exists and the event's value has always a proper value. Relations with occurrence intervals are only allowed to loops (sec. 5.5.2) and selectors (sec. 5.5.3 on the next page), both of which also have access to the value of events. The value can be accessed by the usual dot notation, its identifier is value. More details are explained in the respective sections (sec. 5.5.2 and sec. 5.5.3 on the next page). The distinction whether a relationship exists between the enabling or the occurrence interval is done implicitly: only loop and selector bodies use the occurrence interval, all other intervals use the enabling interval only.

5.5.2. Loops

In Vitruv_L we have one kind of loop. It is controlled by the evaluation of an event's value. We use a head-controlled, negated loop, i.e., we loop until the controlling condition is true, and check the condition before the loop body is activated for the first time.

Loop

Loops have to be declared as elements of classes, their type is Loop. We need the declaration to allow relationships between the loop and other intervals. Each loop depends on an event. To ensure that the loop have access to the event's value, the loop body is started parallel to the event's enabling interval. More precisely, the implicit relation between the loop and the enabling interval is the disjunction of starts, startedby and equals. This implicit relation is required for consistency checking and the semantics of the event-based behavior. It is handled in sec. 7.4.5 on page 146. The loop body is executed until the condition is true or the time-out event occurs. The latter prevents infinite loops, because the loop is at least aborted, if both its body and the enabling interval of its events are finished. But if enabling intervals are infinite, of course infinite loops are also possible.

The body of a loop has a scope of its own. All elements used inside the loop have to be declared in the loop's body, and no relationships to elements outside the loop's

body are allowed. Default elements are alpha and omega, representing beginning and end points of each iteration of the loop's body. As the elements in the loop body may be activated more than once, relationships to elements outside the loop would result in multiple activation of these elements as well, as explained earlier. This means that looping would not be restricted to the loop's body, but is spread out through the entire specification. However, at best this is confusing, hence we do not allow any relationships from outside to the inside of loops. But nevertheless, the entire loop is considered as an interval, the length of which may vary dramatically, and relationships to this interval are possible. This is why loops are declared as elements.

In spec. 5.7 we present an example of a loop, using the button and event declarations of spec. 5.6 on page 71. We have loop l, button b and interval x. At the beginning, button b is activated, which also activates the enabling interval of event pressed. Loop L is repeated until button b is pressed, as denoted by the event b.pressed and its value isPressed. In the body of L, the audio a is declared and started immediately, because a meets alpha, where alpha is the beginning of the loop body. After L finishes, interval x is activated. Implicitly, we have also the relation b.pressed starts or startedBy or equals L between button event and the loop.

Specification 5.7 Loop until the button is pressed.

```
let
  L : Loop;
  b : Button;
  x : Interval;
body
  alpha meets b;
  L loops until (b.pressed.value is isPressed) do
    let
      a : Audio;
    body
      alpha meets a.play;
    end;
  end;
  L meets x;
end;
```

5.5.3. Branching

Branching as a reaction to events means choosing between several alternative *paths of control* depending on the event's value. In contrast to loops, more than one path may exist, but the selected one is not repeated.

paths of control

Selector

Branching is realized – similar to loops – in a special interval type called Selector. Loops and selectors share a set of features. A selector has to be declared as an element and can be used as other elements in the rules. Similar to bodies of loops, the alternative paths are completely encapsulated inside the selector and no relationships exists between them and the outside world. There is also no relationship between the alternative paths. The reason is the same as for loops and events' occurrence intervals: the activation of a certain path is based on an event and any external relationship to its elements would result in an undesirable transfer of the event's nondeterminism characteristic to ordinary intervals. Thus, similar to loop bodies, the alternative paths declare their own set of elements and their behavior is controlled by their own set of rules for each path.

degree of firing

The selector is activated immediately after the occurrence interval of the event, the selector is met by the occurrence interval. The decision process of selecting a path is similar to those used in fuzzy control theory (Yager and Filev, 1994, chapter 4). We have a conditional expression for each path, all of them are evaluated in parallel resulting in fuzzy truth values, which are called the ***degree of firing*** (dof) of each condition (Yager and Filev, 1994, pp. 118-120). After that the expression with the highest dof is selected and the corresponding path is activated. In fuzzy control these rules are usually denoted as IF-THEN-ALSO rules, but we use the form on <Condition> do <Action> to emphasize the aspect of event reaction. If there is more than one path with the same highest dof, a nondeterministic choice is made between them. If the highest dof is 0, i.e. no expression matched, then – as an exception of the beforementioned rule – nothing happens at all. Essentially, this evaluation is a defuzzification process, determining a crisp output of a fuzzy input (Yager and Filev, 1994, p. 121).

Similar to loops, the relations between the event, the entire selector and the selector bodies are only given implicitly in the specification. The entire selector, i.e. the entity declared of type Selector, has the relations starts, startedby or equals to the event enabling. The selector bodies, i.e. the different paths of control, are started immediately after the event occurrence interval and thus have the relation meets to the event occurrence interval. These implicit relations are required for consistency checking and the semantics of the event-based behavior. They are handled in sec. 7.4.5 on page 146.

In spec. 5.8 on the facing page we present a selector working on a button event as defined in spec. 5.6 on page 71. We start with playing video v1. During this, button b is enabled and selector s is activated if the button's occurrence interval is activated. s has two alternatives: if b is pressed, then video v3 is activated, if b is not pressed and thus the TimeOut-value appears, then audio a is played. After the selector finishes, video v2 is activated. Implicitly, we have the relation b.pressed starts or startedBy or equals s.

Specification 5.8 Select between additional video and audio depending on button event.

```
let
  s  : Selector;
  b  : Button;
  v1 : Video;
  v2 : Video;
body
  alpha meets v1;
  b during v1;
  s selects (b.pressed) with rules
      on (isPressed) do
        let v3 : Video;
        body
          alpha starts v3;
        end;
      on (TimeOut) do
        let a : Audio;
        body
          alpha starts a;
        end;
  end;
  s meets v2;
end;
```

5.5.4. Multiple Events and Multiple Reactions

In this section we discuss two complex topics concerning events and their use in loops and selectors: multiple events and multiple reactions.

While explaining loops, it was discernible that during an event's enabling interval the event might occur more than once. If we have at most one event occurring during the enabling interval, loops can only run until this event occurs, if it fulfills the termination condition, or until the following time-out event occurs. This is not sufficient, because we have only two time-points in which the termination condition of the loop is checked. Therefore, we offer a more expressive construct, permitting that during an enabling interval more than one event might occur. These events are called ***multiple events***.

multiple events

However, applying multiple events to selectors is not very useful: consider that two events with same values occur during an enabling interval. The selector would choose for both events the same path due to the same value. This means that the same set of intervals would activated, in particular it is possible that both overlap in rendering a certain media, e.g. a video or an audio. But multiple rendering of the same media results in severe problems: do we render them on the same screen, resulting e.g. in an overlay or mix of videos, or do we have (nondeterministically!) many screens – but what about positioning these screens? Is there the any possibility that the resulting rendering is somehow comprehensible by the user? We think this situation is not satisfying at all, therefore we serialize multiple events: their respective values are held in a queue ordered according to their incoming time.

Loops and selectors work differently on this queue. The termination condition of loops is that either an event with the required value or the time-out event occurs. Thus, whenever the condition is checked, the queue is searched for the respective events. If one is found, the loop terminates. All other events in this queue are ignored and also their ordering. Selectors, however, react only on the first event in the queue, because otherwise the selector's reaction might be unsatisfying as discussed above. Hence, selectors react only on the first event in the queue, all events afterwards are ignored. This ensures that multiple events do not start several selector paths in parallel, which would again make control flow unpredictable.

Events, selectors and loops are declared as elements and are positioned by interval relations. It is possible that one event meets several selectors and loops. In this case, we have ***multiple reactions*** on an event. The aforementioned queues are duplicated for each selector and for each loop, ensuring that they do not interfere with each other. While it might not be very useful to define multiple reactions to an event in its declaring class, it becomes handy if the event is exported by its declaring class. This gains access to the event from the outside of its declaring classes. In this case particularly, it is possible that different classes define selectors and loops operating on the same event. It is useful for classes defining user interface elements declaring events, react on them locally, but also export them to client classes.

multiple reactions

We have seen in the spec. 5.6 on page 71 class Button declaring and exporting event pressed. We have also seen independent event reactions to this event in spec. 5.7 on page 73 and spec. 5.8 on page 75, however, it is not evident that these examples work on the same event or only use different instances of the same class Button resulting in different events. Nevertheless, with a slight modification of the source code it would be possible that both specifications work on the same event as shown in spec. 5.9. Here, class LoopAudio inherits from class SelVideo and declares loop l on the same event b.pressed as the inherited selector s.

Specification 5.9 Multiple reactions to the button event.

```
class SelVideo
  exports b;
  let
    s : Selector;
    b : Button;
    v1, v2 : Video;
  body
    alpha meets v1;
    b during v1;
    s selects (b.pressed) with rules
      ... // see spec. 5.8
    end;
    s meets v2;
  end;
end;

class LoopAudio extends SelVideo
  let
    l : Loop;
    x : Interval;
  body
    alpha meets b;
    l loops until (b.pressed.value is isPressed) do
      ... // see spec. 5.7
    end;
    l meets x;
  end;
end;
```

5.6. Presentations and Scenes

The expressiveness of $Vitruv_L$ presented so far concentrates on specifying multimedia presentations in the small, i.e. how media elements (in $Vitruv_L$ characterized by intervals) relate to each other. But multimedia presentations are also structured in the large. We call the larger structures *scenes*. They have the characteristic that no media element overlaps two scenes and scenes cannot be nested. To make scenes the top-level elements their third characteristic is that no media element is allowed outside scenes. This means that all media elements are parts of scenes and scenes are independent, because they do not share elements. Apparently, scenes are a means for modularization of multimedia presentations in the large, as opposed to classes which structure multimedia presentations in the small.

scenes

Syntactically, scenes are realized by inheriting from class Scene, which itself inherits from class Interval. Class Scene is only a marker class, similar to marker interfaces and classes in Java such as Serializable or Error (Gosling and Arnold, 1996), and does not declare additional elements. Nevertheless, each class inheriting from Scene is interpreted as a scene and is marked as a scene, which explains the wording "marker class".

Scenes are similar to nodes in hypermedia systems (e.g. the Dexter model, see Halasz and Schwartz, 1994), elements and their relations in a scene correspond to the within layer of the Dexter model. We connect scenes to each other with ***links***. We stick here to the simple concept of unidirectional links, more elaborated connections are discussed briefly in sec. 12.2.2 on page 257. Following links in hypertext or hypermedia applications means leaving the origin node completely and moving to a target node. In $Vitruv_L$, this means to abort all activated intervals in the origin scene and to activate the target scene right from the start. Reactivation of a scene does not restore the former state of the scene, i.e. we have neither procedural- nor coroutine-like semantics, but are very close to goto-like jumps. While this is not appealing from a programming language point of view, it is sufficient for coarse grained connection of scenes, more complex behaviors are done inside the scenes.

links

Following links means activating an interval representing a scene. However, since activation of a scene requires leaving the current scene, it is not sufficient to use the scene object as an ordinary interval, because only one relation (meets) makes sense as relation with the scene objects, but all other relations might be used by applying Allen's calculus. Therefore, we introduce the term leave for, which takes a scene object as argument. Every time a scene object s is used somewhere in the body, it must be in the form leave for (s) to underline explicitly that we leave the current scene and go to the new scene. If we activate a scene s, all intervals positioned after scene s in the current scene are not activated. Thus, it is often useful to activate a scene only as a last action or inside a branch of a selector. We should also remark, that it is legal to define relationships between several scenes, but this does not mean that all of them are activated at all.

leave for

In spec. 5.10 on the next page we have two scenes, FrenchCathedrals and Chartres-Details. In the former, we declare element cd with type of the latter scene. Because scenes are not allowed to embed each other, this declaration means only a reference to an object defined elsewhere, in contrast to other element declarations. The value of cd is defined – as usual in Vitruv_L – in the binding and the references are resolved there, too (sec. 5.8.9 on page 93). In the body of FrenchCathedrals, we follow the link to cd and leave behind all currently activated intervals including, in particular, class FrenchCathedrals. Both scenes show a typical pattern. Class FrenchCathedrals defines a menu from which the user can select between different alternatives. One of them is realized in scene ChartresDetails. Hence, element cd is declared locally in the selector. In class ChartresDetails we present some information and finally jump back to menu, i.e. to scene FrenchCathedrals, with leave for (fc).

Please note that in spec. 5.10 on the next page we have a tight connection between the scenes objects even at class level, because the elements are declared as of type FrenchCathedrals and ChartresDetails, respectively. Without loosing functionality it is sufficient to declare the respective elements of type Scene and leave the selection of proper scene instances to the binding, applying late binding. While this is more flexible and decouples both scenes, it might hinder comprehension of the specification.

5.7. The Prelude

We have seen a set of implicitly defined elements of Vitruv_L such as the class Interval, some fuzzy types and others. They all are part of Vitruv_L, and are defined in the *prelude*. The prelude defines a set of standard definitions always available in Vitruv_L. It can be extended by local definitions if needed. Usually, we can omit such a self-written prelude and use the standard prelude. That is always the case if we omit the prelude in the specification. The name prelude is a reminder of SML (Reade, 1989).

prelude

Most important in the prelude are the definitions of standard fuzzy types and their operators. The prelude covers at least the two basic types for truth values and durations, and a set of compound interval relations. The complete standard prelude is defined in appendix A.3.

The fuzzy type for durations was presented in sec. 5.4.2 on page 69. Now we will discuss the truth values. The fuzzy type TRUTH represents fuzzy truth values, e.g. results from comparison of fuzzy sets. The universe is the unit interval $[0.0, 1.0]$. The terms are those of four-valued logic: true, false, unknown and undefined. Additionally we define the standard modifiers very, more_or_less and not. The definition of this type is shown in spec. 5.11 on the next page.

Specification 5.10 Scenes and moving between them.

```
class FrenchCathedrals extends Scene
  let
    menu : Selector;
    ... // some more elements
  body
    ...
    menu selects ...
      on .. do
      let
        cd : ChartresDetails;
      body
        alpha meets leave for (cd); // now move to scene cd
      end;
      ...
    end;
  end;
end;

class ChartresDetails extends Scene
  let
    fc : FrenchCathedrals;
    ...
  body
    ...
    omega isMetBy leave for (fc); // ... and back to the menu
  end;
end;
```

Specification 5.11 A small part of the standard prelude.

```
prelude
  define type TRUTH
    universe [0.0 .. 1.0]
    define term true, false, undefined, unknown;
    define modifier very, more_or_less, not;
  end;
  // further definitions omitted
end;
```

5.8. The Binding

Our modeling concept presented in sec. 4.2.2 on page 53 requires the distinction between concepts and realizations. Declaring classes with intervals and their relationships define the conceptual model. We will now present the *binding*, the realization of the conceptual model.

binding

In contrast to stating constraints, as done in the previously explained parts of Vitruv_L, in the binding we explicitly assign values to fuzzy sets, and functions to modifiers, respectively. The definition of values and functions forms the first part of this section. Afterwards we discuss how the binding occurs in the syntactical structures of Vitruv_L. We introduce contexts as the general mechanism to bind and re-bind values and functions in structures, and apply contexts in the binding of fuzzy types, compound relations, classes and the prelude in this order. Finally, we discuss scenes and the main entry point of our specification defining the start point of our dynamic system specified in Vitruv_L.

5.8.1. L-Values

Assignments of values to identifers are possible in the binding. *L-Values* are those identifiers to which assignments are allowed. It is important to distinguish carefully between identifiers, being l-values, and other identifiers or expressions being values to be assigned. Vitruv_L differs here from most other languages because assignments are not allowed to all accessible variables.

L-Values

The assignment structure of Vitruv_L is leveled and hierarchic. We follow the nesting of objects, fuzzy types and compound relationships and each nesting level assignments are only allowed to attributes defined in this very level. However, all visible elements at each level may be used in the expressions assigned. This strategy ensures that each variable is assigned at most one time and we have only one binding at any time and thus are stateless. This approach is very similar to the binding of DoDL (Doberkat, 1996).

5.8.2. Defining Fuzzy Sets

Each fuzzy set is defined by its membership function. We have two different situations: firstly, for enumerated sets we simply state explicitely for each element its membership value. Secondly, for numeric universes we define the membership function by a (in general piecewise assembled) function from the set's universe to the unit interval. The assembled function consists of a set of linear functions and is introduced by the keyword linear, followed by the ordered set of points consisting of ordered pairs $(x, \mu(x))$. This fuzzy set constructor allows definition of fuzzy sets with an arbitrary linear approximated function without discontinuities. For the sake of eas-

linear

ier computation we restrict Vitruv$_L$ here and do not allow arbitrary functions without linear approximations.

In addition to this general membership function definition, we provide various fuzzy set constructors guaranteeing convexity (def. B.5 on page 280). The most popular convex constructions are the singleton set and the trapezoid, triangle and rectangle shapes (def. B.6 on page 281). Additionally we provide *s*-, *z*-, *sz*- and π-functions which represent the left and the right flank of a bell-shape and their combinations (see def. B.7 on page 281).

All these convex sets are always normalized, i.e. their core is not empty. We do not have to provide the membership values as they are clear from the context (either zero or one). The arguments of the constructors are the required parameters of the function definitions which are at most four numbers for both, the *sz*- and trapezoid-function. Examples of the application of the fuzzy set constructors can be seen in the various code examples in this section.

5.8.3. Defining Modifiers

Modifiers are unary functions taking a fuzzy set as an argument and returning a new fuzzy set. A suitable definition of modifiers depends on the context and application domain. We can apply various definitions of the same modifiers in different contexts, e.g., different definitions of very in different fuzzy types. Therefore, we distinguish the definition of the unary modifier functions from their application context.

Vitruv$_L$ provides a set of predefined non-domain-specific general modifier definitions shown in table A.2 on page 274. However, we need to distinguish the modifier symbol in the specification from the aforementioned modifier definition. Therefore, we collect all available modifier definitions in an entity called Modifier which is syntactically similar to classes and provides an entity for each definition. Access to these definitions is provided by the usual dot-notation, e.g. referring to the definition of not is done by Modifier.not. It implies that the definitions need globally unique names.

Modifier

For the sake of simplicity, domain-specific and more complex modifier definitions can not be added. But in sec. 12.2.1 on page 257, we discuss briefly more elaborated and user definable modifiers as a possible extension to Vitruv$_L$.

We have not discussed the mapping between modifier symbols from the specification to modifier definitions in the binding. Usually, the modifier symbol maps to the definition with the same name, expressing the intuitive matching of symbol and definition: the modifier symbol very is bound to the definition Modifier.very. But it is possible to bind a different definition to the modifier symbol, e.g. very := Modifier.above. The next sections discuss where and how such assignments can be done.

5.8.4. Contexts

We have discussed application-domain and situation dependency of fuzzy sets and modifiers: the fuzzy set describing the concept of a hot temperature or a long duration is always context dependent. In Vitruv_L, the concepts are mostly related to a temporal background (e.g. shortly after). This poses the additional problem that, dependent on the current scale, a concept like “short” might mean 2 seconds regarding the complete presentation or 2 milliseconds regarding lip-synchronization of an audio and a video stream. Therefore, we do not only need to bind values and functions to specification symbols, we also need to re-bind new values or functions to these symbols to achieve context-sensitivity.

In the following, we use the word value also for functions while discussing the binding in general for the sake of simplicity. If the difference between values and functions is important, we state this explicitly.

In Vitruv_L, we have a block structure consisting of classes, fuzzy types and the prelude. In the binding, we assign values to symbols following the block structure of the specification but leaving the class level and moving towards objects: the binding is an equivalent to the run-time environment of programming languages. But in contrast to usual models of run-time environments, such as the contour-model (Waite and Goos, 1984, p. 36), we have to deal with re-binding of values in an inner block to symbols defined in an outer block, but without side-effects. The discussion above concerning the concept of “short” makes it evident that re-bindings need to occur, but are not the same as assignment to global variables in programming languages. We need a controlled area in a symbol’s scope in which we can re-bind its value, but outside this area its value remains unchanged. We call such a controlled area a ***context***[1]. A context is always a range in which new symbols can be bound. These symbols come from the elements of the context’s respective syntactical unit in the specification, e.g. the local declarations in a class definition. According to the syntactical structure, contexts may be nested. An inner context inherits all bindings of its outer context, unless they are hidden by new definitions of symbols inside the context. Additionally, an inherited binding may be re-bound to a new value. The extent of this re-binding is only the current context itself: the binding in the outer context remains unchanged.

context

A more operational view on contexts relies on call-by-values semantics of procedure calls. The effect of using contexts is comparable to a procedure, to which all definitions from outer blocks are passed as parameters with call-by-value. All such definitions are then local copies, which can be modified within the current context, but without effecting the outer contexts. This approach is also known as closure of procedures (Waite and Goos, 1984, p. 73), as used in PROSET (Doberkat et al., 1994).

In the following sections we present the binding of fuzzy types, classes and the prelude. Common to all these elements is that their binding establishes contexts.

[1]Our contexts have nothing to do with the contexts in semantics needed for operational equivalence (Mitchel, 1996, p. 77).

5.8.5. Compound Interval Relations

Compound relations are abstractions similar to procedures or functions in ordinary imperative and functional languages, respectively. Once defined, they can be used throughout the entire specification in various usually very different situations. An important issue of such different situations is that the relative time scale may differ dramatically, as discussed above in the example of concept "short" in the section on contexts (p. 83). Nevertheless we want to apply the same compound relation in such different situations but without defining all possible durations required for the definition of the compound relation. In particular this would demand that each time we apply a compound relation we also need to know how the relation is defined. This would lead the idea of compound relations as abstractions *ad absurdum*!

Therefore, we omit a specific binding of compound relations. Their evaluation relies only on the current bindings of fuzzy type DURATION, because this type is used to denote the duration constraints which are imposed in the definition of compound relations.

5.8.6. Fuzzy Types

The binding of a fuzzy type is an application of the bindings of fuzzy sets and modifiers defined previously. We have to state the type name and assign to each symbol (fuzzy set or modifier) an explicit value.

In specification 5.12 we present the binding of fuzzy type Brightness which was first defined in spec. 5.4 on page 69. The terms are bound to fuzzy sets defined by fuzzy set constructors mentioned above, black and white as trapezoidal fuzzy sets, the other four dark, muddy, shining and bright as triangle fuzzy sets. The graphs of the membership functions are shown in fig. 5.2 on the next page. All modifiers are inherited from the context outside the type, only Modifier.extremely is assigned to the modifier symbol very.

Specification 5.12 Binding of fuzzy type Brightness.

```
in type Brightness
  let black   := trapezoid (0, 0, 10, 20);
  let dark    := triangle (10, 25, 40);
  let muddy   := triangle (25, 40, 120);
  let shining := triangle (100, 170, 190);
  let bright  := triangle (170, 190, 240);
  let white   := trapezoid (230, 240, 255, 255);
  let very    := Modifier.extremely;
end;
```

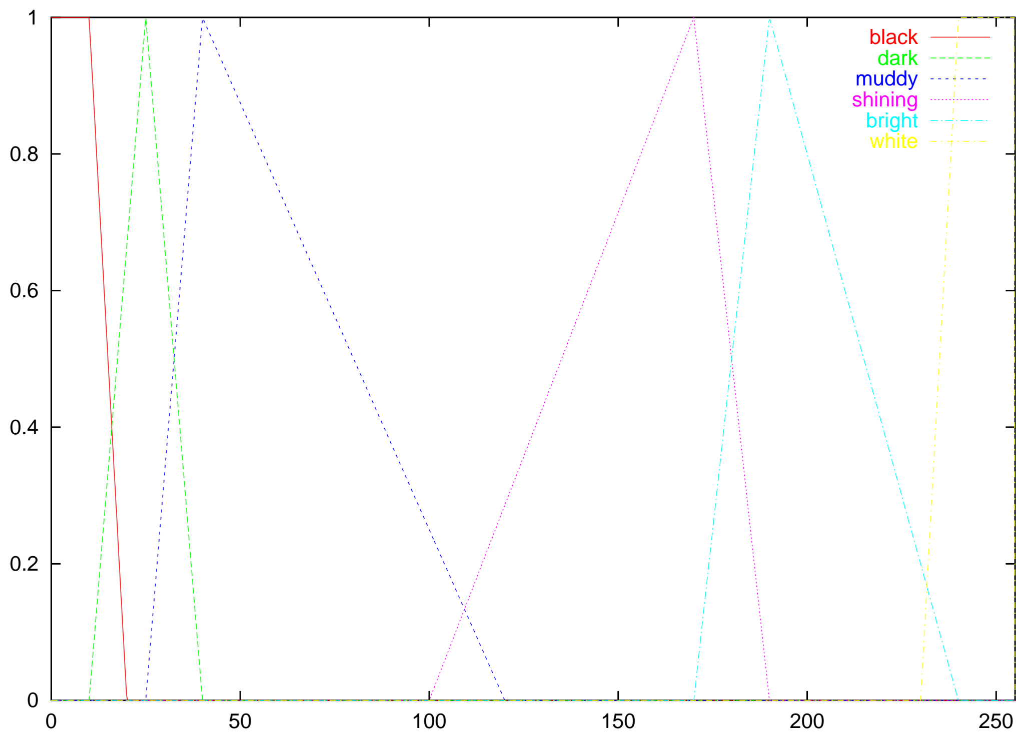

Figure 5.2.: The terms of fuzzy type Brightness (spec. 5.12)

5.8.7. Binding of Classes

The binding of classes is a misleading term, because in the binding we construct and bind only objects and not classes. Nevertheless, the binding is structured by the class of the object which is bound. Therefore, we keep this term.

An object has a object hierarchy (possibly empty), which follows the object's class definition. In the binding, we describe each object as a hierarchy of object bindings. Objects are semantically equivalent to intervals, thus the possible values which are bound to an object are fuzzy sets determining the interval's length or duration. These values can either be given explicitly in the binding or implicitly deduced from the rules in the specification and their respective binding.

For each object, its structure is determined by its class, and each of its sub-objects is individually given. The dominant role of inheritance, structuring the specification of classes, is not repeated in the binding because we focus on objects, and thus the structuring facility of inheritance – the incremental and classified definition – is not useful. Certainly inheritance is essential for polymorphic assignments of an instance of a subclass to a variable with type of the superclass. Therefore, we have to notate both, the object variable and its class which may be a subclass of the variable's specified class.

We can omit the binding for an attribute of a class. As result we gain a nil object. This is particularly important for recursive classes, because it finishes effectively the recursion and therefore ensures finiteness. The special attributes alpha and omega are not allowed as l-values in the binding, according to the discussion in sec. 5.3.2.2 on page 66.

In the following we discuss the binding of objects in various steps and use the four classes A, B, C and D from spec. 5.13 on the next page as examples. We will explain the binding of a simple object, of a simple hierarchy of objects, of inheritance and explain the effect of redefining an outer binding in an inner context.

5.8.7.1. Binding a simple Object

Class A is simple because it does not refer to other classes than class Interval. If we want to bind an object a1 of class A, we have only to consider the element a and its duration constraint. In the simplest case, we refer to the corresponding term of fuzzy type DURATION. This is shown in spec. 5.14 on page 88.

Alternatively, we can assign an explicitly defined fuzzy set as shown in spec. 5.15, here we use the fuzzy set expression triangle (10, 20, 30) which we assign to element a of object a2. In such a situation, it is possible to achieve inconsistent specifications if the bound fuzzy set expression does not correspond to the specification stating that a is large. Therefore, we introduce a local consistency rule for duration constraints.

Type Rule 1 (Consistent Assignment) *Let a be a fuzzy duration with a specified constraint expression c. The value of expression e assigned to a in the binding has to be a*

Specification 5.13 The demo classes A, B, C and D.

```
class A exports a
  let
    a : Interval;
  body
    a is large;
  end;
end;

class B exports d
  let
    ba : A;  // ba is a private element of B
    d : Interval;
  body
    ba meets d;
    d is short;
  end;
end;

class C extends B
  let
    e : Interval;
  body
    e before d;
    e is somewhat (d.length);
  end;
end;

class D
  let
    b : B;
  body
    b is short;
  end;
end;
```

Specification 5.14 Assigning the value of large from DURATION.

```
object a1 : A
  object a : A
    let length := DURATION.large;
  end;
end;
```

Specification 5.15 Assigning an explicite fuzzy set expression.

```
object a2 : A
  object a : A
    let length := triangle (10, 20, 30);
  end;
end;
```

non-fuzzy subset of the value of constraint expression c.

The value of the constraint in this rule is determined by the current binding of type DURATION to which the constraint expression corresponds. In our example the constraint is a is large, therefore the expression triangle (10, 20, 30) has to be a subset of DURATION.large. The rationale of this rule is that such duration constraints form the set of possible values for the duration, and thus each proper value has to be a member of the set of possible values. If we do not assign a crisp duration, we have at least to assign a fuzzy set which is contained in the set of possible values. This requires a subset relationship, which also includes crisp values: they are represented by the singleton set with membership value 1 at the crisp value and zero otherwise. More problematic than crisp values, the empty fuzzy set is always a proper subset of any other fuzzy set, but is certainly no sensible value in our calculus, because it means that no duration is possible at all. We exclude this situation explicitly in the next rule.

Type Rule 2 (Non-empty Assignment) *The value assigned to a duration needs a non-empty support.*

The binding of an object creates a new context. Therefore, we can change inherited values of the outer context. In our case, we could redefine the terms and modifiers of fuzzy type DURATION. In spec. 5.16 on the next page we re-bind the term large of DURATION and afterwards we use the new value to assign it to element a of object a3. The general scheme is firstly to re-bind inherited values and then to bind new values.

Specification 5.16 Re-binding DURATION.

```
object a3 : A
  in type DURATION
    let large := pifunction (0.3, 0.1);
  end;
  object a : A
    let length := DURATION.large;
  end;
end;
```

5.8.7.2. Binding of a Hierarchy

Let us now consider an object b1 of class B which consists of an object ba of class A. The nesting of the objects is repeated in the binding: we start with the outer object b1 of class B, in which we bind the inner object ba needing an assignment to the duration of its element a. After binding ba, we bind the duration of element d of object b1. This is shown in spec. 5.17.

Specification 5.17 Binding a hierarchy of objects

```
object b1 : B
  object ba : A
    object a : A
      let length := DURATION.large;
  end;
  object d : Interval
    let length := DURATION.short;
  end;
end;
```

For simplicity reasons, we assigned only terms of fuzzy type DURATION, but other, more complex expression are also possible. In particular, it is possible to re-bind terms and modifiers of type DURATION in both contexts of b1 and ba independently in the same way, as shown in spec. 5.16.

Object hierarchies have to satisfy the requirement that the duration of objects lower in the hierarchy is not greater than the respective durations of objects higher in the hierarchy. Otherwise, the embedding of objects by a hierarchy is not possible. Certainly, references to scene objects, as needed for jumps, are not considered here.

5.8.7.3. Binding and Inheritance

The binding of an object, which has a class that inherits from other classes, has to reflect the inherited features. Inheritance is a mechanism allowing incremental extensions of a base class and therefore a subclass of such base class has all features of its base class. The binding of object c1 of class C in spec. 5.18 has to consider both the features from class C and B, i.e. e and also d and ba. We cannot simply flatten the classes, because we have private elements in inherited classes which are not visible outside. The inheriting class can introduce new elements with the same name as the inherited private elements. Therefore, we distinguish between exported elements and their private counterparts. The former are addressed as seen above, the latter (but only for inherited ones) are defined recursively with a super-notation, similar to various programming languages. Each super-block consists of assignments to private elements and an additional super-block for the parent class. In spec. 5.18 we have only one super-block for addressing element ba of parent class B. If B inherited from another class, say X, with additional local elements, then we would have a nested super-block for reaching the local elements of the grandfather X of class C.

super

Specification 5.18 Binding of an object with inheritance

```
object c1 : C
  object d : Interval
    let length := DURATION.short;
  end;
  object e : Interval
    let length := DURATION.short;
  end;
  in super
    object ba : A
      object a : A
        let length := DURATION.large;
      end;
    end;
  end;
end;
```

A different situation arises if the specification contains a variable the class of which has descendents. In spec. 5.13 on page 87 we have such a situation in class D, where we have the variable b of class B. We mentioned earlier that polymorphic assignments are possible in the binding and thus we can assign an instance of class C to this variable B. This is shown in spec. 5.19 on the facing page. The variable b of class D is assigned to an object of class C. Syntactically this is expressed by b : C, which does not refer to the specified class B but to class C which inherits from class B.

Specification 5.19 Polymorphic Binding of an Object

```
object d1 : D
  object b : C
    object d : Interval
      let length := DURATION.short;
    end;
    object e : Interval
      let length := DURATION.short;
    end;
    let length := pifunction (10, 20);
    in super
     object ba : A
        let a.length := DURATION.large;
      end;
    end;
  end;
end;
```

5.8.7.4. Loops and Selectors

The binding of loops and selectors is similar to those of classes, because both consists of local declarations and rules. The main difference is that we have several paths in selectors, each of which have their own local declarations and rules. To distinguish between these paths, we use on ... end groups, in which the assignments to locally declared elements of each path can be made. The order of the on ... end groups has to match the order of the path declarations, similar to actual and formal parameters in procedure calls.

In spec. 5.20 on the next page we bind an object of class LoopAudio as specified in spec. 5.9 based upon spec. 5.8 and spec. 5.7. Object a locally declared in the body of loop L is defined below L. The private inherited selector s is handled in the super-block of object la. Each path of s is encapsulated by on and end defining the locally declared elements such as v3 and a.

5.8.8. Binding of the Prelude

The prelude collects definitions of fuzzy types, modifiers and class definitions constituting the basic features of Vitruv_{L}. If omitted in the specification, the standard prelude is used, otherwise a new prelude can be defined. The binding of the prelude reflects this situation: if it is omitted, then the standard binding will be used, otherwise the new valuations are applied.

The binding of the prelude is separated from other parts of the binding by the key-

Specification 5.20 Binding of Loops and Selectors

```
object la : LoopAudio
  in L : Loop
    object a : Audio
      ...
    end
  end;
  object b : Button ... end;
  object x : ... end;
  in super
    object v1 : Video ... end;
    object v2 : Video ... end;
    in s : Selector
      on
        object v3 : Video ... end;
      end;
      on
        object a : Audio ... end;
      end;
    end;
  end;
end;
```

words in prelude ... end. Clearly the prelude opens a new context, but different from ordinary contexts, the valuations given here are also the default values of all others contexts outside of the prelude. The canonical solution would be that the prelude's binding is the outermost binding and all other occur inside the prelude's binding. This matches the informal semantics. But syntactically, this is not appealing.

The binding of the prelude is only seldom given by the specifier, otherwise it would not fulfill its intended task. Therefore, we want to separate the prelude's binding from the binding of the remainder specification, similar to separate compilation units in programming languages. Because of that we define both bindings as siblings at the same level in the syntax tree. However, this mismatch of syntactical and semantical embedding should be no source of confusion.

The binding of the prelude provides default values for the proper binding of the specification itself. But for classes we have the problem that we need defaults on a class and not on an instance level, normally used in the binding. Therefore, we introduce a specific binding which looks similar to those of objects but which starts with class *Ident* instead of the usual object *Ident* : *Ident*. Inside the class binding, anything is exactly as for objects, since conceptually this approach can be seen as creating an anonymous instance of the class under consideration. Of course, we allow at most one binding for a particular class.

5.8.9. Scenes

Elements of type Scene and its sub-types cannot be nested as objects in the object hierarchy as explained in sec. 5.6 on page 78, but their values are only references to scene objects. Consequently, scenes are named objects declared on the top-level of the binding which can be referenced by their name. Each scene is an object hierarchy of its own, i.e. we have a forest of trees together with references in these trees to their roots. We use the operator ref to denote a reference to a scene, the argument of ref is the name of the scene. The scope of ref's arguments is restricted to scenes defined at the top level of the binding to avoid name clashes with elements defined somewhere in the classes.

ref

In spec. 5.21 on the next page we present the binding of scenes FrenchCathedrals and ChartresDetails, declared in spec. 5.10 on page 80. The scene objects are defined at the top level of the binding, and are named fcScene and cdScene. These identifiers are used in the binding of elements cd and fc in classes FrenchCathedrals and ChartresDetails, respectively.

5.8.10. The Main Entry Point

Until now, we have discussed the binding of fuzzy sets, fuzzy types, modifiers, classes, scenes and the prelude. Still missing is the main entry point, defining the initial state

of the system specified. Following Meyer (1992, p. 35), all objects existing in a system have to be accessible transitively from the first object created. Translated into Vitruv_L this means, all objects in a system are contained (or referenced transitively) in a context which is opened by only one object. In this situation the definition of the main entry point becomes apparent: it has to be the first object which is bound immediately after the binding of the prelude. It also has to have type Scene or one its subtypes. The name of the object is not of real interest, nevertheless a sensible name makes the specification more readable. In spec. 5.21 we show the binding of a specification including the prelude and the main entry point init of class Presentation.

Specification 5.21 Binding with main entry point and scenes.

```
bindings
  in prelude
    ... // bindings in the prelude
  end;
  object init : Scene // the main entry point ...
    ... // ... and its values
  end;
  object fcScene : FrenchCathedrals // one well-known scene
    ...
    let cd := ref (cdScene)
  end;
  object cdScene : ChartresDetails // another well-known scene
    ..
    let fc := ref (fcScene);
  end;
end;
```

6. Static Semantics of Vitruv_L

Of course, the concrete grammar given in sec. A.1 on page 265 is not enough to define meaningful Vitruv_L specifications. A specification is only sound if it obeys the syntax, the type rules including scope and visibility rules. We will now establish the static semantics of Vitruv_L, which allows static typing. This includes all informal, implicitly and explicitly given type and scope rules in the language design (sec. 5 on page 59). We follow the style presented by Cardelli (1997).

In the following, we develop the grammar and the respective type rules. We start with some preliminary definitions and then go to declarations of classes, fuzzy types and compound intervals. Next we present the body of classes, and finally we discuss the binding. In each section we refine grammar and type rules. We conclude with some remarks on the approach for the static semantics.

6.1. Introduction

We define the static semantics using a type system which is itself a logical deduction system according to Cardelli (1997), Schmidt (1994) and others. The purpose of the deduction system is to derive whether or not a specification is well-typed, i.e. well-formed regarding the type rules.

The inference rules use judgments of the form $\Gamma \vdash I$, where I is an assertion and its free variables are declared in environment Γ. Operator $\vdash$ is called ***entails*** and states that I can deduced from Γ. Environment Γ is a set of variables, x_i, and their types, θ_i, written as $x_i : \theta_i$, and encloses the free variables of I. Most important for us is the typing judgment which asserts that term M has type θ with respect to environment Γ in which all free variables of M are declared. It has the form:

$\vdash$

$$\Gamma \vdash M : \theta$$

where the colon, :, is a binary predicate. $M : \theta$ can be read as M has type θ, the entire judgment as M has type θ in environment Γ.

:

In complicated judgments we will use parentheses to distinguish between terms to be typed and their type attributes. These parentheses are not part of the language of terms, but should not be confusing for the reader. The above judgment can be written as:

$$\Gamma \vdash (M) : (\theta)$$

Such terms, M, are elements of syntactic domains which correspond to the abstract syntax and are not restricted to nonterminal symbols of the grammar. Because terms of type attributes are also elements of syntactic domains, we can define most terms by a context free grammar.

A particular form of judgments is

$$\Gamma \vdash M$$

where M is again a term. In contrast to $\Gamma \vdash M : \theta$, where we state the term M has type θ, we state here that M is well-formed (i.e. well-typed) with respect to environment Γ. Such judgments are typical for terms not being expression-like and thus not having natural type themselves. However, the term's constituents need to be correctly typed with respect to environment Γ.

A type rule asserts the validity of judgments on the basis of other judgments known to be valid. The syntax of type rules is always such that a single conclusion, $\Gamma \vdash I$, can be drawn from a set of premises, $\Gamma_k \vdash I_k$. A premise can also be a predicate. We notate rules such that premises are written above a line and the conclusion below the line. Additionally a rule may be named. In the following example it is named "sample rule":

$$\text{[sample rule]} \quad \frac{\Gamma_1 \vdash I_1 \quad \ldots \quad \Gamma_n \vdash I_n}{\Gamma \vdash I}$$

If all premises are satisfied, the conclusion must hold. The set of premises may be empty. Similar to natural deduction a derivation in the type system is valid if we can build a proof tree with the judgment of interest as its root. After introducing some rules in the next sections we will present in sec. 6.5 on page 105 an example for a type derivation.

As mentioned above, the terms are defined by a context-free abstract grammar which is in turn defines meta-variables of various syntactical domains. These meta-variables are usually introduced as name of a production. In such cases we do not define the corresponding syntactic domain explicitly, but follow the convention that each domain has the same name as its meta-variable, but written in bold letters. As an example we have the variable *IdList* which takes values from the set ***IdList***, i.e., we have the meta-meta-rule that for each meta-variable v holds:

$$v \in \boldsymbol{v}.$$

The meta-variables can be decorated by super- and subscripts and primes distinguishing various variables from the same domain but generally with different values. If a variable with the same decorations occurs twice in one rule, it always has the same value. Consequently, with different decorations, values need not to be equal.

Literals occurring in terms of the type rules are notated in two different ways: if they are part of the abstract syntax of $\mathsf{Vitruv_L}$, we use the sans-serif font together with

single quotes (e.g. ‘class’); if they are part of the typing attributes, we use a slightly different sans-serif font without quotes (e.g. class).

To ease the incremental definition of the grammar, we allow multiple definitions of the same production *p*. The complete definition of production *p* is the disjunction of all definitions as alternatives. As an example consider the following two definitions of ***p*** :

$$p ::= s_1 \mid s_2$$

$$p ::= s_3$$

The entire definition of *p* is then their combination

$$p ::= s_1 \mid s_2 \mid s_3$$

and the values of *p* are taken from the set ***p***.

6.2. Preliminaries

Throughout the following sections we rely on some basic definitions. We collect them here.

6.2.1. Identifiers and Numerals

Identifiers are elements of the (infinite) set *Id*, ***x*** is an element of *Id*. Identifiers follow the production *Ident* of the concrete grammar in sec. A.1.1.2 on page 265 but with the additional characters ‘#’, used for identifiers of compound relationships, and ‘α’, ‘σ’, and ‘λ’, used in the binding.

Numeric constants are elements of $\mathbb{R} \cup \{\infty, -\infty\}$, following the productions *RealNumber* and *IntegerNumber* (again defined in sec. A.1.1.2 on page 265), ***n*** and ***m*** are such numeric constants.

6.2.2. Type Attributes

The set of type attributes, $\boldsymbol{\Psi}$, has ψ as meta-variable. Type attributes are terms of the type system, that is why they are defined by production rules. In most cases, we reason about well-formed types, but sometimes we have to defer decisions about typing. Thus we introduce two further type attributes partitioning the values of ψ:

$$\psi ::= \theta \mid \varphi$$

where θ and φ denote well-formed and deferred types, respectively.

In our inference rules we often need a typing context or static *environment* Γ . It is

environment Γ

defined as a set of typed identifiers:

$$\Gamma := \{j : \theta_j | j \in J\} \quad J \subset Id \tag{6.1}$$

with the additional constraint

$$(x : \theta_1) \in \Gamma \wedge (x : \theta_2) \in \Gamma \Rightarrow \theta_1 = \theta_2$$

As shorthand notation we use environments sometimes as partial functions:

$$\Gamma(x) := \begin{cases} \theta & \text{if } (x : \theta) \in \Gamma \\ \text{undefined} & \text{else} \end{cases} \tag{6.2}$$

dom
ran

The functions dom and ran take an environment Γ as argument and calculate its domain and range, respectively:

$$\text{dom}(\Gamma) := \{j \,|\, (j : \theta_j) \in \Gamma\}. \tag{6.3}$$

$$\text{ran}(\Gamma) := \{\theta_j \,|\, (j : \theta_j) \in \Gamma\}. \tag{6.4}$$

Other predicates and functions are defined whenever they are used for the first time.

$\theta ::= \tau \mid \Gamma\delta \mid U \mid F \mid \boldsymbol{F} \rightarrow \boldsymbol{F}$

$\tau ::= \mathsf{num} \mid \mathsf{u} \mid \mathsf{f} \mid \mathsf{selector} \mid \mathsf{loop}$

$F ::= U \rightarrow \mathsf{f}$

$U ::= \Gamma \mid [\, n, m \,]$

$\delta ::= \ \mid \mathsf{dec} \mid \mathsf{class} \mid \mathsf{rel} \mid \mathsf{ftype} \mid \mathsf{event} \mid \mathsf{Class} \mid \mathsf{Rel} \mid \mathsf{Ftype}$
$\quad \mid \ \mathsf{Event} \mid \mathsf{val} \mid \mathsf{ref} \mid \mathsf{Sel} \mid \mathsf{Loop}$

Figure 6.1.: Syntax of type attributes

In fig. 6.1 we present the grammar of well-formed type attributes, using Γ as defined in equation (6.1). Their informal meaning is the following: in type τ are simple types, i.e. types of numeric values (num), of elements of enumerated universes of fuzzy types (u), and finally the interval [0.0, 1.0] used for terms and modifiers of fuzzy types (f). Also selector and loop are simple types, they model only declared selectors and loops, which lack any structure. In U we will capture the structure of an universe.

It is either a numeric interval or a type environment, the latter containing the identifiers of the enumerated elements together with their type u. A fuzzy set is a function from an universe U to the interval $[0.0, 1.0]$, thus we introduce function types of the form $\theta_1 \to \theta_2$, denoting the set of all functions from θ_1 to θ_2. We need only a very limited subset of such generic types: fuzzy sets and modifiers. The type F of fuzzy sets is denoted by $U \to \mathsf{f}$. Modifiers are functions between fuzzy sets of the same universe, hence their type is $F \to F$ which is a second-order function.

A type attribute of form $\Gamma\delta$ denotes the various forms of type environments, discriminated by the value of δ. Declaration of fuzzy types, compound relationships, single classes and events have type attribute $\Gamma\mathsf{Ftype}$ (sec. 6.4), $\Gamma\mathsf{Rel}$ (sec. 6.6), $\Gamma\mathsf{Class}$ (sec. 6.7) and $\Gamma\mathsf{Event}$ (sec. 6.7.4), respectively. The lowercase variants ($\Gamma\mathsf{ftype}$, $\Gamma\mathsf{rel}$, $\Gamma\mathsf{class}$, $\Gamma\mathsf{event}$) denote the respective instance, i.e. an identifier x_1 with type attribute $\Gamma\mathsf{Class}$ defines a class, an identifier x_2 with type attribute $\Gamma\mathsf{class}$ is an object of class x_1. A set of declarations, including singleton sets of classes, of fuzzy types, of compound relationships or of elements in a class, result in a type attribute of the form $\Gamma\mathsf{dec}$. With $\Gamma\mathsf{Sel}$ and $\Gamma\mathsf{loop}$ we denote the complex definition of selectors and loops, resp., as they are defined in the class bodies (sec. 6.7.5.4). In the binding (sec. 6.8), we use $\Gamma\mathsf{val}$ and $\Gamma\mathsf{ref}$ to denote assignments of values and references, respectively. For intermediate type environments we have the form Γ, where δ is empty. A usual idiom is that during construction of e.g. a fuzzy type we use the empty δ for intermediate results. The final declaration of the fuzzy type with identifier x is $\{x : \Gamma\mathsf{Ftype}\}\mathsf{dec}$, i.e. a declaration type environment containing the pair of x and the $\Gamma\mathsf{Ftype}$ type attribute as shown in rule 6.32 on page 105.

Additionally, we have some expressions calculating type attributes based on usual set-theoretic operations, e.g. to take the union of type environments. Such expressions are needed for constructing type environments and for use in predicates.

6.2.3. General Building Blocks

6.2.3.1. Well-Formed Environments

An important general judgment assert that an environment is well-formed, which we denote by $\diamond$, and that a type θ is well-formed in environment Γ:

$\diamond$

$$\Gamma \vdash \diamond \tag{6.5}$$

$$\Gamma \vdash \theta \tag{6.6}$$

They come together with the general rule (6.7) that the empty environment $\emptyset$ is well-formed. This rule has no premises and thus is an axiom. In every well-formed environment, the basic types τ are well-formed (6.8). If an identifier x together with its type attribute θ is an element of the type environment, we can derive this type

relationship with rule (6.9).

$$\frac{[\text{empty environment}]}{\emptyset \vdash \diamond} \tag{6.7}$$

$$[\text{primitive types}]\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \tau} \tag{6.8}$$

$$[\text{entail identifier}]\quad \frac{\Gamma \vdash \diamond \quad (\boldsymbol{x} : \theta) \in \Gamma}{\Gamma \vdash (\boldsymbol{x} : \theta)} \tag{6.9}$$

The union of well-formed environments Γ_1 and Γ_2 is well-formed, if the domains of both environments are disjoint.

$$[\text{union of environments}]\quad \frac{\Gamma_1 \vdash \diamond \quad \Gamma_2 \vdash \diamond \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma_1 \cup \Gamma_2 \vdash \diamond} \tag{6.10}$$

6.2.3.2. Numerals

In each well-formed environment, numerals have the type num (remember that $\boldsymbol{n}$ and $\boldsymbol{m}$ are the meta-variables for numerals):

$$[\text{numerals}]\quad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \boldsymbol{n} : \mathsf{num}} \tag{6.11}$$

6.2.3.3. Deferred Types

Sometimes, we have to defer the decision about the type of an identifier. Therefore we introduce a dummy basic type ident, which may be assigned to an identifier. This is formalized in the following grammar and type rule:

$\varphi ::= \mathsf{ident}$

$$[\text{ident intro}]\quad \frac{\Gamma \vdash \diamond \quad \boldsymbol{x} \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \boldsymbol{x} : \mathsf{ident}} \tag{6.12}$$

6.2.3.4. Identifier Lists

A general construct used in several situations is the *IdList*-production, which is a non-empty comma-separated list of distinct identifiers:

$$IdList ::= \boldsymbol{x} \mid IdList \text{ ‘,’ } \boldsymbol{x}$$

Its type attribute is an intermediate type environment Γ collecting the list's identifiers and their types requiring that these type are well-formed because they are elements of $\boldsymbol{\theta}$.

The following rule shows how to construct the type attributes for an identifier list. An initial list with two identifiers is shown in rule (6.13). The identifiers x_1 and x_2 have types θ_1 and θ_2, resp., in the current environment Γ. The type attribute is the set containing both identifiers with their types. In rule (6.14) an existing identifier list *IdList* is extended by identifier x, which have type θ and which must not already occur in *IdList*.

[IdList with two identifiers]

$$\frac{\Gamma \vdash x_1 : \theta_1 \quad \Gamma \vdash x_2 : \theta_2 \quad x_1 \neq x_2}{\Gamma \vdash (x_1 \text{ ‘,’ } x_2) : \{x_1 : \theta_1, x_2 : \theta_2\}} \tag{6.13}$$

[append IdList]

$$\frac{\Gamma \vdash x : \theta \quad \Gamma \vdash \mathit{IdList} : \Gamma_1 \quad x \notin \mathrm{dom}(\Gamma_1)}{\Gamma \vdash (\mathit{IdList} \text{ ‘,’ } x) : \Gamma_1 \cup \{x : \theta\}} \tag{6.14}$$

These rules work well if we have a list of at least two elements. For the singleton case, we need always an extra rule when we use an *IdList* in another production. As an example we present these two rules for the export list of a class (compare the rules (6.48) and (6.49)). The first one deals with the singleton case, the second one with at least two elements in the list.

$$\frac{\Gamma \vdash x : \theta}{\Gamma \vdash \text{‘exports’ } x \text{ ‘;’} : \{x : \theta\}\mathsf{dec}} \qquad \frac{\Gamma \vdash \mathit{IdList} : \Gamma_1}{\Gamma \vdash \text{‘exports’ } \mathit{IdList} \text{ ‘;’} : \Gamma_1\mathsf{dec}}$$

A more obvious rule would convert a single identifier x with type θ into an *IdList* with type $\{x : \theta\}$. But this would result in an endless recursion by choosing the wrong path at the nondeterministic decision whether a single identifier is an *IdList* or not, as shown in the following derivation:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash x : \theta}{\Gamma \vdash x : \{x : \theta\}}}{\Gamma \vdash x : \{x : \{x : \theta\}\}}}{\vdots}$$

6.2.3.5. Dot-Notation

The dot-notation (I) requires a recursive descendent through nested type assignments.

$$I ::= \boldsymbol{x} \mid I \text{ `.' } \boldsymbol{x}$$

We have to distinguish between the different kinds of environments. If in an expression of the form $I.x$ the type attribute of I is a type environment Γ, but not for a class, then we require that x is in the domain of Γ. If I results in a class, we only allow access to exported elements of this class, which we extract via $\Gamma_1(\text{`exports'})$ (cf. sec. 6.7.1 on page 110).

$$\text{[ident expr]} \quad \frac{\Gamma \vdash \boldsymbol{I} : \Gamma_1\delta \quad \delta \neq \mathsf{class} \quad \Gamma_1 \vdash \boldsymbol{x} : \theta}{\Gamma \vdash \boldsymbol{I} \text{ `.' } \boldsymbol{x} : \theta} \tag{6.15}$$

$$\text{[ident expr class]} \quad \frac{\Gamma \vdash \boldsymbol{I} : \Gamma_1\mathsf{class} \quad \Gamma_1(\text{`exports'}) \vdash \boldsymbol{x} : \theta}{\Gamma \vdash \boldsymbol{I} \text{ `.' } \boldsymbol{x} : \theta} \tag{6.16}$$

6.3. The Entire Specification

The entire specification, S, consists of the prelude, P, the declaration, D, and the binding B. The prelude and the declaration build an environment in which the binding must be well-formed. We have the following meta-variables:

S	specification
B	binding
P	prelude
D	declaration
C	class definition
$FType$	fuzzy type definition
$IntDef$	interval relationship definition

and the abstract grammar is defined as follows:

$$S ::= P\ D\ B$$

$$D ::= \mid C \mid FType \mid IntDef \mid D_1\ D_2$$

$$P ::= \mid \text{`prelude'}\ D\ \text{`end'}\ \text{`;'}$$

The specification and the binding do not have types, but need to be well-formed. Thus we introduce new judgments stating the specification and the binding are well-typed with respect to a type environment Γ:

$$\Gamma \vdash B \tag{6.17}$$

$$\Gamma \vdash S \tag{6.18}$$

The type rules for the declarations are straight-forward, because the main work is done in the constituents, defined in the following sections. We demand that declarations do not declare the same identifier twice, thus the intersection of the two type environments' domain has to be empty. The declarations in the first declaration, Γ_1, may be used in the second one, thus we take the union of Γ_1 with environment Γ obtaining the environment for typing the second declaration. The prelude consists of a declaration, hence its type attribute is the declaration's type environment.

[combine declarations]

$$\frac{\Gamma \vdash D_1 : \Gamma_1\mathsf{dec} \quad \Gamma \cup \Gamma_1 \vdash D_2 : \Gamma_2\mathsf{dec} \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash D_1\ D_2 : (\Gamma_1 \cup \Gamma_2)\mathsf{dec}} \tag{6.19}$$

[the prelude]

$$\frac{\Gamma \vdash D : \Gamma_1\mathsf{dec}}{\Gamma \vdash \text{`prelude'}\ D\ \text{`end' `;'} : \Gamma_1\mathsf{dec}} \tag{6.20}$$

Specifications are well-formed if, starting from an empty environment, the declaration builds an environment in which the binding is well-formed.

[specification]

$$\frac{\emptyset \vdash P : \Gamma\mathsf{dec} \quad \Gamma \vdash D : \Gamma_1\mathsf{dec} \quad \Gamma_1 \vdash B}{\emptyset \vdash P\ D\ B} \tag{6.21}$$

In the following we discuss the different forms of declarations, fuzzy types (sec. 6.4), compound relations (sec. 6.6 on page 107), and classes (sec. 6.7 on page 109). Finally we discuss the binding (sec. 6.8 on page 119).

6.4. Fuzzy Types

A fuzzy type introduces a universe, and several terms and modifiers. Elements in a class can be instances of fuzzy types. Fuzzy values, consisting of an expression of terms and modifiers, can be assigned to such elements. The grammar of a fuzzy type definition is as follows:

$FType$ ::= 'define' 'type' $\boldsymbol{x}$ FU FT FM 'end' ';'

FU ::= 'universe' FUL ';'

FUL ::= '[' n '..' m ']' ';' | '{' $IdList$ '}' ';'

FT ::= 'define' 'term' $IdList$ ';'

FM ::= | 'define' 'modifier' $IdList$ ';'

The type attribute describing a fuzzy type is an environment $\Gamma\mathsf{Ftype}$, containing all terms and modifiers with their types and the universe definition. The universe is encoded as a special element of the environment where the identifier is the reserved word 'universe'.

We start with the type rules for an enumerated universe, i.e. for the list of identifiers. We modify the type of an identifier from ident to u and collect all listed identifiers in an environment Γ_1.

$$\text{[Ident universe intro]} \quad \frac{\Gamma \vdash x : \mathsf{ident}}{\Gamma \vdash x : \mathsf{u}} \tag{6.22}$$

$$\text{[singleton universe]} \quad \frac{\Gamma \vdash x : \mathsf{u}}{\Gamma \vdash (\text{'universe' '\{' } x \text{ '\}' ';'}) : \{\text{'universe'} : \{x : \mathsf{u}\}\}} \tag{6.23}$$

$$\text{[enumerated universe]} \quad \frac{\Gamma \vdash \mathit{IdList} : \Gamma_1 \quad \mathrm{ran}(\Gamma_1) = \{\mathsf{u}\}}{\Gamma \vdash (\text{'universe' '\{' } \mathit{IdList} \text{ '\}' ';'},) : \{\text{'universe'} : \Gamma_1\}} \tag{6.24}$$

The rule for the closed interval universe of a fuzzy type follows:

$$\text{[interval universe]} \quad \frac{\Gamma \vdash n : \mathsf{num} \quad \Gamma \vdash m : \mathsf{num} \quad n \leq m}{\Gamma \vdash (\text{'universe' } n \text{ '..' } m \text{ ';'}) : \{\text{'universe'} : [n,m]\}} \tag{6.25}$$

Terms and modifiers are very similar, only the type attribute differs. We use F to denote a fuzzy set, and $F \to F$ to denote a unary function between two fuzzy sets. We start again with *IdList* for the typing rules but this time we convert ident to F and $F \to F$, resp. In rule (6.28) and in rule (6.31) we require that the elements of Γ_1 are only terms and modifiers, resp.:

$$\text{[Ident fuzzy set intro]} \quad \frac{\Gamma \vdash x : \mathsf{ident} \quad \Gamma \vdash \text{'universe'} : U}{\Gamma \vdash x : U \to \mathsf{f}} \tag{6.26}$$

[define singleton term]

$$\frac{\Gamma \vdash x : F}{\Gamma \vdash (\text{`define' `term'}\ x\ \text{`;'}) : \{x : F\}} \tag{6.27}$$

[define terms]

$$\frac{\Gamma \vdash \mathit{IdList} : \Gamma_1 \quad \mathrm{ran}(\Gamma_1) = \{F\}}{\Gamma \vdash (\text{`define' `term'}\ \mathit{IdList}\ \text{`;'}) : \Gamma_1} \tag{6.28}$$

The rules for modifiers are symmetric:

[Ident modifier intro]

$$\frac{\Gamma \vdash x : \mathsf{ident} \quad \Gamma \vdash \text{`universe'} : U}{\Gamma \vdash x : (U \to \mathsf{f}) \to (U \to \mathsf{f})} \tag{6.29}$$

[define singleton modifier]

$$\frac{\Gamma \vdash x : F \to F}{\Gamma \vdash (\text{`define' `modifier'}\ x\ \text{`;'}) : \{x : F \to F\}} \tag{6.30}$$

[define modifiers]

$$\frac{\Gamma \vdash \mathit{IdList} : \Gamma_1 \quad \mathrm{ran}(\Gamma_1) = \{F \to F\}}{\Gamma \vdash (\text{`define' `modifier'}\ \mathit{IdList}\ \text{`;'}) : \Gamma_1} \tag{6.31}$$

The definition of fuzzy type x results in a type environment of identifier x the type attribute of which is the union of three fuzzy type environments. Terms and modifiers rely on the universe, and all modifiers are not allowed to be already used as a term. As mentioned above, the pseudo identifier 'universe', used for the universe, is a reserved word and thus cannot be an element of the terms or modifiers. To obtain easier rules later on, we assign to the universe a fuzzy set ($F = U \to \mathsf{f}$) in Γ_4 (6.33). This is the rule for defining a fuzzy type:

[define fuzzy type]

$$\frac{\emptyset \vdash \mathit{FU} : \Gamma_1 \quad \Gamma_1 \vdash \mathit{FT} : \Gamma_2 \quad \Gamma_1 \cup \Gamma_2 \vdash \mathit{FM} : \Gamma_3}{\Gamma \vdash (\text{`define' `type'}\ x\ \mathit{FU}\ \mathit{FT}\ \mathit{FM}\ \text{`end'`;'}) : (\{x : (\Gamma_4 \cup \Gamma_2 \cup \Gamma_3)\mathsf{Ftype}\}\mathsf{dec})} \tag{6.32}$$

where

$$\Gamma_4 = \{\text{`universe'} : \Gamma_1(\text{`universe'}) \to \mathsf{f}\} \tag{6.33}$$

6.5. Example of a Type Derivation

After presenting the type rules for fuzzy types in the last section, we can now show an example of a type derivation, i.e. a proof tree. Our examples is the definition of a fuzzy type Temp:

```
define type Temp
  universe [ 0.0 , 100.0 ];        //   = FU
   define term cold , hot ;         //   = FT
  define modifier very , not ;     //   = FM
end ;
```

We want to prove what is stated in rule (6.32). Because our input is too long, we break up the proof tree into four parts, the typing hypothesis and the three sub-terms of the hypothesis.

Let us start with the hypothesis, the root of our proof tree:

$$\frac{\emptyset \vdash FU : \Gamma_1 \quad \Gamma_1 \vdash FT : \Gamma_2 \quad \Gamma_1 \cup \Gamma_2 \vdash FM : \Gamma_3}{\Gamma \vdash \text{'define type Temp'}\ FU\ FT\ FM\ \text{'end'';'} : (\{\text{'Temp'} : (\Gamma_4 \cup \Gamma_2 \cup \Gamma_3)\mathsf{Ftype}\}\mathsf{dec})}$$

where

$$\Gamma_4 = \{\text{'universe'} : \Gamma_1(\text{'universe'}) \rightarrow \mathsf{f}\}$$

Now we have to prove that the expanded productions *FU*, *FT* and *FM* are well-typed and we will show how their type attributes Γ_1, Γ_2 and Γ_3 are constructed. A new fuzzy type defines a new and fresh scope for its constituents, thus the environment for the production is empty ($\emptyset$).

The proof-tree for *FU* follows rule (6.25)

$$\frac{\dfrac{\emptyset \vdash \mathbf{0.0} : \mathsf{num} \quad \emptyset \vdash \mathbf{100.0} : \mathsf{num} \quad \mathbf{0.0} \leq \mathbf{100.0}}{\emptyset \vdash \text{'0.0 .. 100.0'} : [\mathbf{0.0}, \mathbf{100.0}]}}{\emptyset \vdash \text{'universe 0.0 .. 100.0;'} : \{\text{'universe'} : [\mathbf{0.0}, \mathbf{100.0}]\}}$$

and we get as result the type attribute $\Gamma_1 = \{\text{'universe'} : [\mathbf{0.0}, \mathbf{100.0}]\}$.

For *FT* we follow rule (6.28):

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash \diamond \quad \text{'cold'} \notin \mathrm{dom}(\Gamma_1)}{\Gamma_1 \vdash \text{'cold'} : \mathsf{ident}}}{\Gamma_1 \vdash \text{'cold'} : U \rightarrow \mathsf{f}} \qquad \dfrac{\dfrac{\Gamma_1 \vdash \diamond \quad \text{'hot'} \notin \mathrm{dom}(\Gamma_1)}{\Gamma_1 \vdash \text{'hot'} : \mathsf{ident}}}{\Gamma_1 \vdash \text{'hot'} : U \rightarrow \mathsf{f}}}{\Gamma_1 \vdash (\text{'cold, hot'}) : \{\text{'cold'} : U \rightarrow \mathsf{f}\} \cup \{\text{'hot'} : U \rightarrow \mathsf{f}\}}}{\Gamma_1 \vdash (\text{'define term cold, hot;'}) : \{\text{'cold'} : U \rightarrow \mathsf{f}, \text{'hot'} : U \rightarrow \mathsf{f}\}}$$

and we get type attribute $\Gamma_2 = \{\text{'cold'} : U \rightarrow \mathsf{f}, \text{'hot'} : U \rightarrow \mathsf{f}\}$.

Finally for *FM* we follow rule (6.31), which is symmetric to the tree above for *FT*. For simplicity we use $\Gamma_4 = \Gamma_1 \cup \Gamma_2$.

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma_4 \vdash \diamond \quad \text{'very'} \notin \mathrm{dom}(\Gamma_4)}{\Gamma_4 \vdash \text{'very'} : \mathsf{ident}}}{\Gamma_4 \vdash \text{'very'} : (U \rightarrow \mathsf{f}) \rightarrow (U \rightarrow \mathsf{f})} \qquad \dfrac{\dfrac{\Gamma_4 \vdash \diamond \quad \text{'not'} \notin \mathrm{dom}(\Gamma_4)}{\Gamma_4 \vdash \text{'not'} : \mathsf{ident}}}{\Gamma_4 \vdash \text{'not'} : (U \rightarrow \mathsf{f}) \rightarrow (U \rightarrow \mathsf{f})}}{\Gamma_4 \vdash (\text{'very, not'}) : \{\text{'very'} : (U \rightarrow \mathsf{f}) \rightarrow (U \rightarrow \mathsf{f})\} \cup \{\text{'not'} : (U \rightarrow \mathsf{f}) \rightarrow (U \rightarrow \mathsf{f})\}}}{\Gamma_4 \vdash (\text{'define modifier very, not;'}) : \{\text{'very'} : (U \rightarrow \mathsf{f}) \rightarrow (U \rightarrow \mathsf{f}), \text{'not'} : (U \rightarrow \mathsf{f}) \rightarrow (U \rightarrow \mathsf{f})\}}$$

These three derivations construct the four needed type environments:

$$\Gamma_1 = \{\text{'universe'} : [0.0, 100.0]\}$$
$$\Gamma_2 = \{\text{'cold'} : U \to \mathsf{f}, \text{'hot'} : U \to \mathsf{f}\}$$
$$\Gamma_3 = \{\text{'very'} : (U \to \mathsf{f}) \to (U \to \mathsf{f}), \text{'not'} : (U \to \mathsf{f}) \to (U \to \mathsf{f})\}$$
$$\Gamma_4 = \{\text{'universe'} : U \to \mathsf{f}\}$$

where

$$U = \Gamma_1(\text{'universe'}) = [0.0, 100.0]$$

which were used in the root of our complete proof tree. We have shown how to apply the rules to prove that the declaration of Temp is well typed. During this proof we constructed the type attribute of the fuzzy type declaration which is needed if we want to prove other parts of the specification using Temp.

6.6. Compound Interval Relationships

The definition of compound interval relationships is similar to class definitions because in both situations we have a set of local class instances and a set of rules defining constraints on these instances. Therefore, we defer the detailed discussion of the rules to the discussion of the class bodies.

$IntDef$ ::= $IntHead$ '=' $LetInt$ $Rules$ ';'

$IntHead$ ::= 'define' 'interval' 'relation' x_1 $IdSeq$ x_2

$IdSeq$ ::= x | $IdSeq_1$ $IdSeq_2$

$LetInt$ ::= 'let' $IdList$ 'in'

$Rules$ ::= 'rules' $Rule$ 'end'

$Rule$ ::= $IntRel$ | $Constraint$ | $Rule_1$ $Rule_2$

The type attribute of a compound interval relation is a type environment, $\Gamma\mathsf{Rel}$, containing the two arguments of the relations (see rule (6.42). In this respect a compound relation resembles binary functions or better a procedure with two arguments. As usual in such situations, the argument declaration is considered as a declaration block of its own (Schmidt, 1994, p. 94), accounting for the type attribute used here.

The identifier sequence takes the rather longish name of compound interval relations. The type of an identifier sequence is ident. The whitespace between parts of

identifier sequences is not suitable to be used in domains of type environments, because we need there ordinary identifiers. Thus, in rule (6.34) we use the symbol '#', not occurring in production *Ident* of the concrete grammar, to concatenate the non-whitespace parts of the identifier sequence. This results in ordinary identifiers in the type systems, because '#' occurs in the symbol set of ***Ident***, as defined in sec. 6.2 on page 97. This procedure should not irritate the reader, it is only needed for technical reasons.

$$[\text{ident sequence}] \quad \frac{\Gamma \vdash \mathit{IdSeq}_1 : \mathsf{ident} \quad \Gamma \vdash \mathit{IdSeq}_2 : \mathsf{ident} \quad (\mathit{IdSeq}_1\text{'\#'}\mathit{IdSeq}_2) \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\mathit{IdSeq}_1\text{'\#'}\mathit{IdSeq}_2) : \mathsf{ident}} \tag{6.34}$$

Each identifier in the *LetInt* refers implicitly to the standard class 'Interval', which needs to be defined in the current environment Γ, i.e. we need $(\text{'Interval'} : \Gamma_1\mathsf{Class}) \in \Gamma$ as produced by rule (6.67).

$$[\text{Ident interval intro}] \quad \frac{\Gamma \vdash \text{'Interval'} : \Gamma_1\mathsf{Class} \quad \Gamma \vdash x : \mathsf{ident}}{\Gamma \vdash x : \Gamma_1} \tag{6.35}$$

We explicitly state in the following rules that all identifiers in the *LetInt*-production have the type of class 'Interval' and are not used in environment Γ.

$$[\text{singleton let interval}] \quad \frac{\Gamma \vdash x : \Gamma_1 \quad \Gamma \vdash \text{'Interval'} : \Gamma_1\mathsf{Class} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\text{'let'}\ x\ \text{'in'}) : \{x : \Gamma_1\}\mathsf{dec}} \tag{6.36}$$

$$[\text{let IdList intervals}] \quad \frac{\Gamma \vdash \mathit{IdList} : \Gamma_1 \quad \Gamma \vdash \text{'Interval'} : \Gamma_2\mathsf{Class} \quad \mathrm{ran}(\Gamma_1) = \{\Gamma_2\} \quad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_1) = \emptyset}{\Gamma \vdash (\text{'let'}\ \mathit{IdList}\ \text{'in'}) : \Gamma_1\mathsf{dec}} \tag{6.37}$$

The head of the interval declaration combines the two formal arguments with the identifier of the relation. The formal arguments get the type of class 'Interval', independent of whether or not they have a different type attribute in Γ, and need to be distinct. The identifier of the relation needs to be fresh, hence the requirement that its type attribute is ident.

$$[\text{interval head}] \quad \frac{\Gamma \vdash \text{'Interval'} : \Gamma_1\mathsf{Class} \quad x_1 \neq x_2 \quad \Gamma \vdash \mathit{IdSeq} : \mathsf{ident}}{\Gamma \vdash (x_1\ \mathit{IdSeq}\ x_2) : \{\mathit{IdSeq} : \{x_1 : \Gamma_1, x_2 : \Gamma_1\}\}\mathsf{dec}} \tag{6.38}$$

The rules in a compound interval relationship do not create any values, thus an expression-like type attribute is not appropriate. We introduce a new judgment asserting that a rule ***Rule*** is well-formed in a type environment Γ:

$$\Gamma \vdash \textit{\textbf{Rule}} \tag{6.39}$$

With this judgment we can give the typing rule for a sequence of rules.

$$\text{[combine rules]} \quad \frac{\Gamma \vdash \textit{\textbf{Rule}}_1 \quad \Gamma \vdash \textit{\textbf{Rule}}_2}{\Gamma \vdash \textit{\textbf{Rule}}_1 \textit{\textbf{Rule}}_2} \tag{6.40}$$

As stated above, the typing rules for constraints and interval rules are given in later section when discussing the body of classes.

The typing environment for the rules of compound interval relationships consists of the local interval declarations unified with the relationship's formal arguments. The scoping rules of $\mathsf{Vitruv_L}$ allow that local declarations override definitions from the outside. Thus, we need the asymmetric union operator $\sqcup$:

$\sqcup$

$$\Gamma_1 \sqcup \Gamma_2 := \Gamma_2 \cup (\Gamma_1 \setminus \{(\boldsymbol{x} : \theta_1) | (\boldsymbol{x} : \theta_1) \in \Gamma_1 \wedge \exists \theta_2 : (\boldsymbol{x} : \theta_2) \in \Gamma_2\}) \tag{6.41}$$

Now we can build the complete definition of the compound interval relation. The ***LetInt*** production declares variables in the local scope of the interval relation, thus its type environment contains only the definition of class 'Interval' and the formal arguments of the relation. The rules may refer to definitions outside the interval relation, but local definitions override outer ones, hence we use environment $\Gamma \sqcup (\Gamma_1 \cup \Gamma_3)$.

$$\text{[compound interval definition]} \quad \frac{\begin{array}{cc} \Gamma \vdash \textit{\textbf{IntHead}} : \{\boldsymbol{x} : \Gamma_1\}\mathsf{dec} & \Gamma \vdash \text{'}\mathsf{Interval}\text{'} : \Gamma_2\mathsf{Class} \\ \{\text{'}\mathsf{Interval}\text{'} : \Gamma_2\mathsf{Class}\} \cup \Gamma_1 \vdash \textit{\textbf{LetInt}} : \Gamma_3\mathsf{dec} & \Gamma \sqcup (\Gamma_1 \cup \Gamma_3) \vdash \textit{\textbf{Rule}} \end{array}}{\Gamma \vdash (\textit{\textbf{IntHead}}\ \text{'='}\ \textit{\textbf{LetInt}}\ \text{'}\mathsf{rules}\text{'}\textit{\textbf{Rule}}\ \text{'}\mathsf{end}\text{'}\ \text{';'}) : \{\boldsymbol{x} : \Gamma_1\mathsf{Rel}\}\mathsf{dec}} \tag{6.42}$$

6.7. Classes

A class declares a set of elements, either variables of a fuzzy type, instances of classes, events, loops and selectors. These elements may be exported, i.e. they are made visible outside the class. All elements can be used in the body, where we state rules on the elements. Additionally, a class A can inherit definitions from another class *B*, i.e. all exported elements of this class *B* are available in A.

C ::= 'class' $\boldsymbol{x}$ $\mathit{Inherits}$ $\mathit{Exports}$ $\mathit{Letpart}$ Body 'end' ';'

$\mathit{Inherits}$ ::= | 'extends' I

$Exports$ ::= | ‘exports’ $IdList$ ‘;’

$Letpart$ ::= ‘let’ $DLet$

$DLet$ ::= | x_1 ‘:’ x_2 ‘;’ | $LoopLet$ | $SelLet$ | $EventLet$ | $DLet_1$ $DLet_2$

$LoopLet$ ::= x ‘:’ ‘Loop’ ‘;’

$SelLet$ ::= x ‘:’ ‘Selector ‘;’

$EventLet$ ::= x_1 ‘:’ ‘Event’ ‘[’ x_2 ‘]’ ‘;’

$Body$ ::= ‘body’ R ‘end’ ‘;’

Later, we will continue the grammar definition for the body of classes.

6.7.1. Representing of a Class

A class is a complex structured entity. In various situations we need different parts of its type structure notably for defining the export, for sub-classing and in the binding. The distinction between local and exported elements has to be reflected in the typing: exported elements can be used outside the defining class, i.e. in other classes, thus they are part of the interface of a class. In contrast to that the local elements are only usable inside the class, but we need them again in the binding of the class, hence we need information about local elements also in the type structure. Therefore, we use a structured type environment for classes which consists of three elements:

1. identifier ‘extends’ is typed with the super class.
2. identifier ‘exports’ is typed with a type environment for the exported elements of the class including the inherited ones.
3. identifier ‘let’ is typed with a type environment for the private elements of the class. In this type environment we embed the inherited private elements again with identifier ‘let’: Consider class C_1 and C_2 where C_1 inherits from C_2 with private elements Γ_1 and Γ_2, resp. Then the private elements of C_2 are embedded in Γ_1:
$$(\text{‘let’} : \Gamma_2) \in \Gamma_1.$$

The three identifiers from above are reserved words of the concrete grammar, and thus there cannot be any clashes with user-defined identifiers of elements in classes.

6.7.2. Sub-Classing and Inheritance

Classes enjoy a sub-class relation defined by inheritance. All classes inherit directly or indirectly from class 'Interval', except 'Interval' itself. In the type system, we use a transitive and reflexive variant of the sub-classing relationship, denoted by $<:$, to obtain easier rules than with an irreflexive relationship. We capture sub-classing with a new judgment

$<:$

$$\Gamma \vdash \Gamma_2\mathsf{Class} <: \Gamma_1\mathsf{Class} \tag{6.43}$$

which asserts that class x_2 is a sub-class of x_1 if $x_1 : \Gamma_1\mathsf{Class}$ and $x_2 : \Gamma_2\mathsf{Class}$. The transitivity of sub-classing is given in the next rule, followed by reflexivity.

$$[\text{transitivity of sub-classing}]\quad \frac{\Gamma \vdash \Gamma_1\mathsf{Class} <: \Gamma_2\mathsf{Class} \quad \Gamma \vdash \Gamma_2\mathsf{Class} <: \Gamma_3\mathsf{Class}}{\Gamma \vdash \Gamma_1\mathsf{Class} <: \Gamma_3\mathsf{Class}} \tag{6.44}$$

$$[\text{reflexivity of sub-classing}]\quad \frac{\Gamma \vdash \boldsymbol{x} : \Gamma_1\mathsf{Class}}{\Gamma \vdash \Gamma_1\mathsf{Class} <: \Gamma_1\mathsf{Class}} \tag{6.45}$$

The inheritance relation is defined via keyword 'extends' (6.46). If it is omitted, then the class inherits implicitly from class 'Interval'. This is covered in rule (6.62).

$$[\text{inherit explicit}]\quad \frac{\Gamma \vdash \boldsymbol{I} : \Gamma_1\mathsf{Class}}{\Gamma \vdash (\text{'extends'}\ \boldsymbol{I}) : \Gamma_1\mathsf{Class}} \tag{6.46}$$

We can deduce sub-classing from the type structure of class, as defined in detail in sec. 6.7.5 on page 113.

$$[\text{deduce sub-classing}]\quad \frac{\Gamma \vdash x_1 : \Gamma_1\mathsf{Class} \quad \Gamma \vdash x_2 : \Gamma_2\mathsf{Class} \quad \Gamma_1(\text{'extends'}) = \Gamma_2\mathsf{Class}}{\Gamma \vdash \Gamma_1\mathsf{Class} <: \Gamma_2\mathsf{Class}} \tag{6.47}$$

6.7.3. Exports

The export clause lists all elements which are exported. So we have simply to collect them. If the export clause is empty, i.e. ***Exports*** is the empty word ϵ, then its type attribute is the empty set, i.e. $\emptyset\mathsf{dec}$.

$$[\text{exports IdList}]\quad \frac{\Gamma \vdash \boldsymbol{IdList} : \Gamma_1}{\Gamma \vdash (\text{'exports'}\ \boldsymbol{IdList}\ \text{';'}) : \Gamma_1\mathsf{dec}} \tag{6.48}$$

$$[\text{exports singleton}]\quad \frac{\Gamma \vdash \boldsymbol{x} : \theta}{\Gamma \vdash (\text{'exports'}\ \boldsymbol{x}\ \text{';'}) : \{\boldsymbol{x} : \theta\}\mathsf{dec}} \tag{6.49}$$

6.7.4. Local Declarations

The let-part of a class declares local elements. The typing rules are straightforward. If we declare an element x_1 of class or fuzzy type x_2, then the type attribute of x_1 will be same as for x_2, but we change the corresponding uppercase discriminator to its lowercase counterpart (e.g. Class to class in rule (6.50)) to distinguish types from their instances. Event declarations define the type of the value-field, which has to be a fuzzy type. That is why the event's type attribute, defined in rule (6.52), is an environment containing the pair of identifier value and its type attribute, constructed according to (6.51). Selectors and loops are basic types, hence they require only a well-formed environment Γ. Their structure is revealed in the rules of the body and we update the type attributes of selectors and loops after type checking the rules.

[declare class instance element]
$$\frac{\Gamma \vdash x_2 : \Gamma_1 \mathsf{Class}}{\Gamma \vdash (x_1 `\text{:}' x_2 `\text{;}') : \{x_1 : \Gamma_1 \mathsf{class}\}} \tag{6.50}$$
[declare fuzzy type element]
$$\frac{\Gamma \vdash x_2 : \Gamma_1 \mathsf{Ftype}}{\Gamma \vdash (x_1 `\text{:}' x_2 `\text{;}') : \{x_1 : (\Gamma_1 \mathsf{ftype}\}} \tag{6.51}$$
[declare event element]
$$\frac{\Gamma \vdash x_2 : \Gamma_1 \mathsf{Ftype}}{\Gamma \vdash (x_1 `\text{:}' \text{ `}\mathsf{Event}\text{' `[' } x_2 \text{`]' `;'}) : \{x_1 : (\{\text{`}\mathsf{value}\text{'} : \Gamma_1 \mathsf{ftype}\}\mathsf{Event})\}} \tag{6.52}$$
[declare selector element]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash (x `\text{:}' \text{ `}\mathsf{Selector}\text{' `;'}) : \{x : \mathsf{selector}\}} \tag{6.53}$$
[declare loop element]
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash (x `\text{:}' \text{ `}\mathsf{Loop}\text{' `;'}) : \{x : \mathsf{loop}\}} \tag{6.54}$$

We use the asymmetric union of type environments Γ and Γ_1 to prove the typing of $DLet_2$ in rule (6.55). Finally, in rule (6.56) we have completed the definition of local elements. Here, we require that the locally declared elements do not redefine an global identifier defined elsewhere. We need this to ensure that in particular inherited elements are not redefined, but it also makes no sense to redefine identifers of classes, fuzzy types or compound relationships. If productionDLet is the empty word ϵ, then its type attribute is the empty set, i.e. $\emptyset\mathsf{dec}$.

[add local elements]
$$\frac{\Gamma \vdash DLet_1 : \Gamma_1 \quad \Gamma \sqcup \Gamma_1 \vdash DLet_2 : \Gamma_2 \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash (DLet_1 DLet_2) : (\Gamma_1 \cup \Gamma_2)} \tag{6.55}$$

[letpart]

$$\frac{\Gamma \vdash \textit{DLet} : \Gamma_1 \quad \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma_1) = \emptyset}{\Gamma \vdash (\text{`let'}\textit{DLet}) : \Gamma_1\mathsf{dec}} \tag{6.56}$$

6.7.5. The Class Definition

The type rules for the class definition has to consider three cases:

1. classes inheriting explicitly from a class, rule (6.57)
2. classes inheriting implicitly class from Interval, rule (6.62)
3. the special class Interval, rule (6.67)

The difference between these three variants is the handling of inherited elements, but the remaining parts of the rules for class definitions are the same. Therefore, we use the same identifiers in all three rules to highlight differences and similarities.

The identifier of the class to type check is x. The type attribute of the inherited class is $\Gamma_1\mathsf{Class}$. We need to determine all elements accessible in the body of our class x and we have to divide these elements into private and exported ones. The set of all elements (Γ_5) is the union of inherited ($\Gamma_1(\text{`exports'})$) and locally declared elements (Γ_3) together with the current class itself and the pseudo-variable 'this'. Exported elements are those we inherit and the local elements listed in the export-clause, the remaining elements are private. The private elements contain the private elements of the inherited class ($\Gamma_1(\text{`let'})$), we need them in the binding section. The body has attribute Γ_7 containing the structures of selectors and loops which have to be propagated into Γ_3 and Γ_4. Now we have all elements of the type environment Γ_6 of our class x: the three elements for the inheritance relation, for the exported and for the local elements, respectively.

This is the first rule for classes which inherit explicitly from other classes.

[general class definition]

$$\frac{\begin{array}{ccc}\Gamma \vdash \textit{Inherits} : \Gamma_1\mathsf{Class} & \Gamma_2 \vdash \textit{Letpart} : \Gamma_3\mathsf{dec} & \Gamma_3 \vdash \textit{Exports} : \Gamma_4\mathsf{dec} \\ \Gamma_5 \vdash \textit{Body} : \Gamma_7 & x \notin \mathrm{dom}(\Gamma) & x \neq \text{`Interval'}\end{array}}{\Gamma \vdash (\text{`class'}\ x\ \textit{Inherits Exports Letpart Body}\ \text{`;'}) : \{x : \Gamma_6\mathsf{Class}\}\mathsf{dec}} \tag{6.57}$$

where

$$\Gamma_2 = \Gamma \sqcup \Gamma_1(\text{`exports'}) \sqcup \{x : \Gamma_6\mathsf{Class}, \text{`this'} : \Gamma_6\} \tag{6.58}$$

$$\Gamma_5 = \Gamma_2 \sqcup \Gamma_3 \tag{6.59}$$

$$\begin{aligned}\Gamma_6 = \{&\text{`extends'} : \Gamma_1\mathsf{Class}, \text{`exports'} : (\Gamma_1(\text{`exports'}) \cup \Gamma_8), \\ &\text{`let'} : (((\Gamma_3 \sqcup \Gamma_7) \setminus \Gamma_8) \cup \{\text{`this'} : \Gamma_6, \text{`let'} : \Gamma_1(\text{`let'})\})\mathsf{dec}\}\end{aligned} \tag{6.60}$$

$$\Gamma_8 = \{x : \theta | x \in \mathrm{dom}(\Gamma_4) \wedge (x : \theta) \in (\Gamma_3 \sqcup \Gamma_7)\} \tag{6.61}$$

Classes without explicit inheritance seem to be roots of the inheritance relation, but they inherit implicitly from ‘Interval’. We have only two changes to the rule above: The *Inherits* is omitted and the type attribute $\Gamma_1\mathsf{Class}$ is that of class ‘Interval’.

$$\frac{\begin{array}{ccc}\Gamma \vdash \text{‘Interval’} : \Gamma_1\mathsf{Class} & \Gamma_2 \vdash \mathit{Letpart} : \Gamma_3\mathsf{dec} & \Gamma_3 \vdash \mathit{Exports} : \Gamma_4\mathsf{dec} \\ \Gamma_5 \vdash \mathit{Body} : \Gamma_7 & x \notin \mathrm{dom}(\Gamma) & x \neq \text{‘Interval’}\end{array}}{\Gamma \vdash (\text{‘class’}\ x\ \mathit{Exports}\ \mathit{Letpart}\ \mathit{Body}\ \text{‘;’}) : \{x : \Gamma_6\mathsf{Class}\}\mathsf{dec}}\ [\text{root class definition}] \tag{6.62}$$

where

$$\Gamma_2 = \Gamma \sqcup \Gamma_1(\text{‘exports’}) \sqcup \{x : \Gamma_6\mathsf{Class}, \text{‘this’} : \Gamma_6\} \tag{6.63}$$
$$\Gamma_5 = \Gamma_2 \sqcup \Gamma_3 \tag{6.64}$$
$$\begin{aligned}\Gamma_6 = \{&\text{‘extends’} : \Gamma_1\mathsf{Class}, \text{‘exports’} : (\Gamma_1(\text{‘exports’}) \cup \Gamma_8), \\ &\text{‘let’} : (((\Gamma_3 \sqcup \Gamma_7) \setminus \Gamma_8) \cup \{\text{‘this’} : \Gamma_6, \text{‘let’} : \Gamma_1(\text{‘let’})\})\mathsf{dec}\}\end{aligned} \tag{6.65}$$
$$\Gamma_8 = \{x : \theta | x \in \mathrm{dom}(\Gamma_4) \wedge (x : \theta) \in (\Gamma_3 \sqcup \Gamma_7)\} \tag{6.66}$$

Finally, we have the class definition of ‘Interval’ which is somewhat easier, because we can drop all references to the super-class, i.e. $\Gamma_1 = \emptyset$.

$$\frac{\begin{array}{ccc}\Gamma_2 \vdash \mathit{Letpart} : \Gamma_3\mathsf{dec} & \Gamma_3 \vdash \mathit{Exports} : \Gamma_4\mathsf{dec} & \\ \Gamma_5 \vdash \mathit{Body} : \Gamma_7 & x \notin \mathrm{dom}(\Gamma) & x = \text{‘Interval’}\end{array}}{\Gamma \vdash (\text{‘class’}\ x\ \mathit{Exports}\ \mathit{Letpart}\ \mathit{Body}\ \text{‘;’}) : \{x : \Gamma_6\mathsf{Class}\}\mathsf{dec}}\ [\text{class Interval definition}] \tag{6.67}$$

where

$$\Gamma_2 = \Gamma \sqcup \{x : \Gamma_6\mathsf{Class}, \text{‘this’} : \Gamma_6\} \tag{6.68}$$
$$\Gamma_5 = \Gamma_2 \sqcup \Gamma_3 \tag{6.69}$$
$$\Gamma_6 = \{\text{‘extends’} : \emptyset\mathsf{Class}, \text{‘exports’} : \Gamma_8, \text{‘let’} : (((\Gamma_3 \sqcup \Gamma_7) \setminus \Gamma_8) \cup \{\text{‘this’} : \Gamma_6\})\mathsf{dec}\} \tag{6.70}$$
$$\Gamma_8 = \{x : \theta | x \in \mathrm{dom}(\Gamma_4) \wedge (x : \theta) \in (\Gamma_3 \sqcup \Gamma_7)\} \tag{6.71}$$

6.7.5.1. The Body of a Class

The final step is the application of fuzzy types and elements of that type in expressions which we find in the body of a class. We discuss interval constraints, interval relationships and event reaction in this order. The grammar for the body is the following, where the *binop*-production refers to the binary operators ‘and’, ‘or’, ‘intersect’ and ‘union’:

$R ::= \ \mid \mathit{ConsRule} \mid \mathit{IntRel} \mid \mathit{Loop} \mid \mathit{Selector} \mid \boldsymbol{R_1}\ \boldsymbol{R_2}$

$\mathit{ConsRule} ::= \mathit{Constraint}$ ‘;’

$\mathit{Constraint} ::= I$ ‘is’ E

$E ::= I \mid E_1\ \mathit{binop}\ E_2 \mid \boldsymbol{I}$ ‘(’ E ‘)’

$\mathit{IntRel} ::= \mathit{Int}_1\ \mathit{Rel}\ \mathit{Int}_2$ ‘;’

$\mathit{Int} ::= I \mid$ ‘leave’ ‘for’ ‘(’ I ‘)’

$\mathit{Rel} ::= \mathit{IdSeq} \mid \mathit{Rel}_1$ ‘or’ Rel_2

$\mathit{Loop} ::= x$ ‘loops’ ‘until’ ‘(’ EC ‘)’ ‘do’ $\mathit{Letpart}\ \mathit{Body}$ ‘end’ ‘;’

$EC ::= EI$ ‘is’ E

$EI ::= I$ ‘.’ ‘value’

$\mathit{Selector} ::= x$ ‘selects’ ’(’ I ‘)’ ‘with’ ‘rules’ $\mathit{SelRule}$ ‘end’ ‘;’

$\mathit{SelRule} ::= \mathit{SelAnte}\ \mathit{SelCons}$ ‘;’ $\mid \mathit{SelRule}\ \mathit{SelRule}$

$\mathit{SelAnte} ::=$ ‘on’ E

$\mathit{SelCons} ::=$ ‘do’ $\mathit{Letpart}\ \mathit{Body}$

Loops and selectors can only be defined once, hence we have to do some bookkeeping. While rules in the body are not expression which have a particular type, we misuse a type environment for the bookkeeping. Every rule results in a type environment Γ, but only loops and selectors fill some values into Γ, all other rules result in an empty type environment $\Gamma = \emptyset$. This is explained in detail in sec. 6.7.5.4 on page 117. All other rules result in empty environments. Combining rules unifies the respective type environments, but as usual their domains have to be disjoint. The entire body propagates the type environment of its rules (6.73). If there are no rules in the body (i.e. production R reduces to the empty word), the type environment of the body is empty.

[combine rules in body]

$$\frac{\Gamma \vdash \boldsymbol{R_1} : \Gamma_1 \quad \Gamma \vdash \boldsymbol{R_2} : \Gamma_2 \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash \boldsymbol{R_1 R_2} : \Gamma_1 \cup \Gamma_2} \tag{6.72}$$

[body]

$$\frac{\Gamma \vdash \boldsymbol{R} : \Gamma_1}{\Gamma \vdash \text{‘begin’}\ \boldsymbol{R}\ \text{‘end’ ‘;’} : \Gamma_1} \tag{6.73}$$

6.7.5.2. Interval Constraints

Interval constraints assign fuzzy sets to elements of classes. These fuzzy sets are results of expressions. If the target of an interval constraint or an expression is not a fuzzy type but a class instance, we use as default the element 'length', defined in class 'Interval' and thus always available.

[apply modifier]

$$\frac{\Gamma \vdash E : F \quad \Gamma \vdash I : F \rightarrow F}{\Gamma \vdash I \text{`('} E \text{`)'} : F} \tag{6.74}$$

[apply binop]

$$\frac{\Gamma \vdash E_1 : F \quad \Gamma \vdash E_2 : F}{\Gamma \vdash (E_1 \, \textit{binop} \, E_2) : F} \qquad \textit{binop} \in \{\text{`and', `or', `intersect', `union'}\} \tag{6.75}$$

[default element length]

$$\frac{\Gamma \vdash E : \Gamma_1\text{class} \quad \Gamma_1 \vdash \text{`exports'} : \Gamma_2\text{dec} \quad \Gamma_2 \vdash \text{`length'} : F}{\Gamma \vdash E : F} \tag{6.76}$$

[Constraint]

$$\frac{\Gamma \vdash I : \Gamma_1\text{ftype} \quad \Gamma_1 \vdash \text{`universe'} : F \quad \Gamma_1 \vdash E : F}{\Gamma \vdash I \text{`is'} \, E \, : F} \tag{6.77}$$

[Constraint with default element]

$$\frac{\Gamma \vdash I : \Gamma_1\text{class} \quad \Gamma_1(\text{`exports'}) \vdash \text{`length'} : \Gamma_3\text{ftype} \quad \Gamma_3 \vdash \text{`universe'} : F \quad \Gamma_3 \vdash E : F}{\Gamma \vdash I \text{`is'} \, E \, : F} \tag{6.78}$$

[Constraint rule]

$$\frac{\Gamma \vdash \textit{Constraint} : F}{\Gamma \vdash \textit{Constraint} \text{`;'} : \emptyset} \tag{6.79}$$

6.7.5.3. Interval Relationships

Interval relationships are binary predicates the arguments of which are instances of class 'Interval'. Therefore, we require that the type of the formal arguments is that of class 'Interval' (6.80) and that the actual arguments conform to this type (6.82), i.e. $\Gamma_1 <: \Gamma_4$ and $\Gamma_2 <: \Gamma_4$.

[IdSeq intro]

$$\frac{\Gamma \vdash \textit{IdSeq} : \Gamma_1\text{Rel} \quad \{\Gamma(\text{`Interval'})\} = \text{ran}(\Gamma_1)}{\Gamma \vdash \textit{IdSeq} : \Gamma_1} \tag{6.80}$$

[add Rel]

$$\frac{\Gamma \vdash \mathit{Rel}_1 : \Gamma_1 \quad \Gamma \vdash \mathit{Rel}_2 : \Gamma_2 \quad \mathrm{ran}(\Gamma_1) = \mathrm{ran}(\Gamma_2)}{\Gamma \vdash (\mathit{Rel}_1 \text{ `or' } \mathit{Rel}_2) : \Gamma_1 \cup \Gamma_2} \tag{6.81}$$

[IntRel]

$$\frac{\begin{array}{lll}\Gamma \vdash \mathit{Rel} : \Gamma_3 & \Gamma \vdash \mathit{Int}_1 : \Gamma_1\mathsf{class} & \Gamma \vdash \mathit{Int}_2 : \Gamma_2\mathsf{class} \\ \mathrm{ran}(\Gamma_3) = \{\Gamma_4\} & \Gamma_1 <: \Gamma_4 & \Gamma_2 <: \Gamma_4\end{array}}{\Gamma \vdash \mathit{Int}_1\ \mathit{Rel}\ \mathit{Int}_2 \text{ `;' } : \emptyset} \tag{6.82}$$

The intervals used arguments for interval relationships can either be ordinary intervals, i.e. instances of classes, or leave for-expressions. The latter require arguments conforming to class Scene.

[leave for]

$$\frac{\Gamma \vdash \text{`Scene'} : \Gamma_1\mathsf{Class} \quad \Gamma \vdash I : \Gamma_2\mathsf{class} \quad \Gamma_2 <: \Gamma_1}{\Gamma \vdash \text{`leave' `for' `(' } I \text{ `)'} : \Gamma_2\mathsf{class}} \tag{6.83}$$

6.7.5.4. Loops and Selectors

Loops and selectors are similar to classes, because they have their own let-parts and bodies. All elements used in the bodies have to be declared in the respective let-part and are in particular not allowed to be elements of the loops' or selectors' surrounding classes. Hence, we need to reduce the type environment Γ used for typing any rule in the body of a class. We need to filter out all type instances of Γ, i.e. we get only type definitions. This filter is called *filterTypes* and is defined in (6.84).

filterTypes

$$\mathit{filterTypes}(\Gamma) := \{x : \Gamma_1\delta | (x : \Gamma_1\delta) \in \Gamma \wedge \delta \in \{\mathsf{Class}, \mathsf{Ftype}\}\} \tag{6.84}$$

The type attributes for loops and selectors is a nested type environment, where each new declaration block (only one for loops and possibly several for selectors) is encoded as a separate type environment (see (6.87) and (6.95)). Each of these environments are attributes of identifier on, which cannot be used as an ordinary identifier because it is a reserved word. The structure propagation applies to selectors and loops as well, therefore we use the asymmetric union for the definition of Γ_4 in rules (6.89) and (6.92). The order of such blocks in selectors is preserved by the nesting of the type environments. This is similar to the nesting of private variables in the type environment for classes.

To guarantee that a loop refers only to an event and not to arbitrary fuzzy constraints as termination condition, we use event constraints (6.86), whose type environment is extended for the TimeOut-value.

[Event Interval]

$$\frac{\Gamma \vdash I : \Gamma_1\mathsf{event} \quad \Gamma_1 \vdash \text{`value'} : \Gamma_2\mathsf{ftype}}{\Gamma \vdash I \text{ `.' `value'} : \Gamma_2\mathsf{ftype}} \tag{6.85}$$

[Event Constraint]

$$\frac{\Gamma \vdash \mathit{EI} : \Gamma_1\mathsf{ftype} \quad \Gamma_1(\text{‘universe’}) = F \quad \Gamma_1 \cup \{\text{‘TimeOut’} : F\} \vdash E : F}{\Gamma \vdash \mathit{EI}\ \text{‘is’}\ E : F} \tag{6.86}$$

[loop]

$$\frac{\Gamma \vdash x : \mathsf{loop} \quad \Gamma \vdash \mathit{EC} : F \quad \Gamma_1 \vdash \mathit{DLet} : \Gamma_2 \quad \Gamma_1 \sqcup \Gamma_2 \vdash \mathit{Body} : \Gamma_3}{\Gamma \vdash x\ \text{‘loops’ ‘until’ ‘(’}\mathit{EC}\text{‘)’ ‘do’ ‘let’}\ \mathit{DLet}\ \mathit{Body}\ \text{‘;’ ‘end’ ‘;’} : \{x : \Gamma_4\mathsf{Loop}\}} \tag{6.87}$$

with

$$\Gamma_1 = \mathit{filterTypes}(\Gamma) \tag{6.88}$$
$$\Gamma_4 = \{\text{‘on’} : \Gamma_2 \sqcup \Gamma_3\} \tag{6.89}$$

All expressions in the antedecents of selectors have to conform to the event value’s fuzzy type F extended by the TimeOut value (6.90). We encode the type of pseudo-identifier on (6.95) as the value’s fuzzy type F. The consequent consists of a declaration block and a set of rules, similar to loop bodies. Again, we have to filter out all type instances in Γ to ensure that no references to elements declared outside the selector are made (6.91). Selector rules and the combination is straightforward (see. (6.93) and (6.94)).

[selector antedecent]

$$\frac{\Gamma \vdash \text{‘on’} : \Gamma_1\mathsf{ftype} \quad \Gamma_1 \vdash \text{‘universe’} : F \quad \Gamma_1 \cup \{\text{‘TimeOut’} : F\} \vdash E : F}{\Gamma \vdash \text{‘on’}\ E : F} \tag{6.90}$$

[selector consequent]

$$\frac{\Gamma \vdash \mathit{DLet} : \Gamma_1 \quad \Gamma_2 = \mathit{filterTypes}(\Gamma) \quad \Gamma_2 \sqcup \Gamma_1 \vdash \mathit{Body} : \Gamma_3}{\Gamma \vdash \text{‘do’ ‘let’}\ \mathit{DLet}\ \mathit{Body}\ \text{‘;’} : \Gamma_4} \tag{6.91}$$

$$\text{where } \Gamma_4 = \Gamma_1 \sqcup \Gamma_3 \tag{6.92}$$

[selector rule]

$$\frac{\Gamma \vdash \mathit{SelAnte} : F \quad \Gamma \vdash \mathit{SelCons} : \Gamma_1}{\Gamma \vdash \mathit{SelAnte}\ \mathit{SelCons}\ \text{‘;’} : \Gamma_1} \tag{6.93}$$

[combine selector rules]

$$\frac{\Gamma \vdash \mathit{SelRule}_1 : \Gamma_1 \quad \Gamma \vdash \mathit{SelRule}_2 : \Gamma_2}{\Gamma \vdash \mathit{SelRule}_1 \mathit{SelRule}_2 : \{\Gamma_1 \cup \{\text{‘on’} : \Gamma_2\}\}} \tag{6.94}$$

[selector]

$$\frac{\begin{array}{cc}\Gamma \vdash x : \mathsf{selector} & \Gamma \vdash I : \Gamma_1\mathsf{event} \\ \Gamma_1 \vdash \text{‘value’} : \Gamma_2\mathsf{ftype} & \Gamma \cup \{\text{‘on’} : \Gamma_2\mathsf{ftype}\} \vdash \mathit{SelRule} : \Gamma_3\end{array}}{\Gamma \vdash x\ \text{‘selects’ ‘(’}I\text{‘)’ ‘with’ ‘rules’}\ \mathit{SelRule}\ \text{‘end’ ‘;’} : \{x : \{\text{‘on’} : \Gamma_3\}\mathsf{Sel}\}} \tag{6.95}$$

6.8. The Binding Section

In the binding section we assign values from and to structures defined in the declaration part of the specification. Our focus lies on checking whether or not the values assigned fit the declared structures. The effect of the bindings is captured in the evaluation semantics of $\mathsf{Vitruv}_\mathsf{L}$, presented later in sec. 7 on page 131.

We start with the grammar of the binding section:

B ::= 'bindings' $inPrelude$ $BindSpec$ 'end' ';'

$inPrelude$ ::= | 'in' 'prelude' $PList$ 'end' ';'

$PList$ ::= $inType$ | $inClass$ | $PList_1$ $PList_2$

$BindSpec$ ::= $TopLevelObject$ | $BindSpec$ $BindSpec$

$TopLevelObject$ ::= $inObject$

6.8.1. Representing the Binding

The type attributes of the binding reflects the bookkeeping needed to check the validity of creation and assignments to objects and fuzzy types. Notably we have to deal with assigned values, references to top-level objects, l-values and the main entry point of the specification.

6.8.1.1. Value Assignments

While we have mostly structured assignments in the binding, traditionally not resulting in type attributes, we supply each of them with a type environment, $\Gamma\mathsf{val}$, collecting all identifiers with assigned values. We require this for bookkeeping reasons, because we do not allow more than one assignment to each variable. Alternatively, as the semantics of multiple assignments is not clear, we would need some kind of resolution strategy such as the last assignment wins, or there is an order of assignments with side-effects as in imperative languages. But these alternatives are not satisfactory at all and thus we forbid multiple assignments.

6.8.1.2. References to Scenes

Bookkeeping is also needed for references, because they refer to objects defined as top-level objects. This is realized as part of the value type environments as an additional environment $\Gamma\mathsf{ref}$, containing name and type of the reference (6.133). While checking the entire binding, we have to check that all these references only refer to the top level objects. All top level objects and their types are collected in an type environment of pseudo identifier σ (6.112). We need this special identifier, because the top

level objects' identifiers have a $\Gamma\mathsf{val}$ type attribute, but we need a $\Gamma\mathsf{Class}$ attribute for checking the references, hence this indirection. For a convenient dealing with the type environment of top level objects, we define two operators $\cup_\sigma$ (6.96) and $\cap_\sigma$ (6.99):

$\cup_\sigma$
$\cap_\sigma$

$$\Gamma_1 \cap_\sigma \Gamma_2 := \Gamma_3 \cap \Gamma_4 \tag{6.96}$$

where

$$\Gamma_3 := \Gamma_1 \setminus \Gamma_1(\text{`}\sigma\text{'}) \tag{6.97}$$
$$\Gamma_4 := \Gamma_2 \setminus \Gamma_2(\text{`}\sigma\text{'}) \tag{6.98}$$

and

$$\Gamma_1 \cup_\sigma \Gamma_2 := \Gamma_3 \cup \Gamma_4 \tag{6.99}$$

where

$$\Gamma_3 := (\Gamma_1 \setminus \Gamma_1(\text{`}\sigma\text{'})) \cup (\Gamma_2 \setminus \Gamma_2(\text{`}\sigma\text{'})) \tag{6.100}$$
$$\Gamma_4 := \{\text{`}\sigma\text{'} : (\Gamma_1(\text{`}\sigma\text{'}) \cup \Gamma_2(\text{`}\sigma\text{'}))\} \tag{6.101}$$

Intersection $\cap_\sigma$ considers everything except the environments for top level objects. Union $\cup_\sigma$ handles both parts independently: Γ_3, unifying everything except the environments for top level objects, and Γ_4. The latter is a type environment containing identifier σ and its type attribute, which is the unification of the top level objects' type attributes found in Γ_1 and Γ_2.

Checking the integrity of references and top level objects requires identifying all top level objects and all references in a value type environment. References are leaves in the type environment, i.e. we do a recursive descend through the value environment, the top level objects are collected as type environment for special identifier σ. We define predicate *cr* (6.102) checking the references by splitting its argument into top level objects and values, and applying predicate *checkrefs* (6.103). These predicates requires that for each identifier's value type environment the references are valid, which is checked by applying *checkref* (6.104), using *checkrefs* again for the recursive descend. If we find a reference $\{x : \Gamma_2\}\mathsf{ref}$, then type Γ_2 must conform to that of top level object x, which is $\Gamma_1(x)$.

$$cr(\Gamma) := checkrefs(\Gamma(\text{`}\sigma\text{'}), \Gamma \setminus \Gamma(\text{`}\sigma\text{'})) \tag{6.102}$$

$$checkrefs(\Gamma_1, \Gamma_2) := \bigwedge_{x \in \mathrm{dom}(\Gamma_2)} checkref(\Gamma_1, \Gamma_2(x)) \tag{6.103}$$

$$checkref(\Gamma_1, \theta) := \begin{cases} checkrefs(\Gamma_1, \Gamma) & \text{if} \quad \theta = \Gamma\mathsf{val} \\ \Gamma_2\mathsf{Class} <: \Gamma_1(x) & \text{if} \quad \theta = \{x : \Gamma_2\}\mathsf{ref} \\ \text{true} & \text{else} \end{cases} \tag{6.104}$$

6.8.1.3. L-Values

L-values are identifiers to which assignments are possible. As stated in sec. 5.8.1 on page 81, we need to distinguish between identifiers, being l-values, and other identifiers or expressions being values to be assigned, because the scoping rules for assignments differ from rules for visibility. In general, assignments are only possible for elements in the innermost environment but not for those elements defined in outer environments. The details are carried out in the type rules of terms *VAssign*, *inType*, *inObject* and *inSuper*. We introduce a special element, λ, in our type environments collecting all l-values and their type attributes.

λ

6.8.1.4. The main Entry Point

We denote the main entry point in the specification in the type environment of production *BindSpec* with α. Each top level object's type environment is duplicated with α as additional identifier (6.108).

α

6.8.2. The Entire Binding

Now we can establish the rules for the entire binding applying the aforementioned encodings.

The rules for prelude's binding ((6.106) and (6.107)) are straightforward applying the rules for objects and types, forbidding multiple assignments. Top level objects are possible candidates for the main object, hence we introduce α in rule (6.108). In (6.109) we propagate only the left α of each pair, guaranteeing that the first top level object becomes the main entry object of the specification.

Finally, in rule (6.105) we present the entire binding occurring here in its long form in the conclusion as judgment according to rule (6.17). We add the type environment $\Gamma_\lambda = \{`\lambda' : \emptyset\}$ to ensure that in all other dependent type environments the extraction of $\Gamma(`\lambda')$ is successful. We require that all references to top level objects are valid by checking Γ_2 with predicate *cr*. Requirement $\mathrm{dom}(\Gamma_1) \cap_\sigma \mathrm{dom}(\Gamma_2) = \emptyset$ in rule (6.109) implies that $\Gamma_1(`\sigma')$ and $\Gamma_2(`\sigma')$ are disjoint according to rule (6.112), therefore the application of $\cup_\sigma$ in the conclusion of rule (6.109) is valid.

$$\text{[bindings]}\quad \frac{\Gamma \cup \{`\lambda' : \emptyset\} \vdash \textit{inPrelude} : \Gamma_1\mathsf{val} \quad \Gamma \cup \{`\lambda' : \emptyset\} \vdash \textit{BindSpec} : \Gamma_2\mathsf{val} \quad \textit{cr}(\Gamma_2)}{\Gamma \vdash (\text{`bindings'}\ \textit{inPrelude}\ \textit{BindSpec}\ \text{`end'}\ \text{`;'})} \tag{6.105}$$

$$\text{[prelude]}\quad \frac{\Gamma \vdash \textit{PList} : \Gamma_1\mathsf{val}}{\Gamma \vdash (\text{`in'}\ \text{`prelude'}\ \textit{PList}\ \text{`end'}\ \text{`;'}) : \Gamma_1\mathsf{val}} \tag{6.106}$$

[combine PList]

$$\frac{\Gamma \vdash \textit{PList}_1 : \Gamma_1\mathsf{val} \quad \Gamma \vdash \textit{PList}_2 : \Gamma_2\mathsf{val} \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash \textit{PList}_1\textit{PList}_2 : (\Gamma_1 \cup \Gamma_2)\mathsf{val}} \tag{6.107}$$

[top level object]

$$\frac{\Gamma \vdash \textit{inObject} : \Gamma_1\mathsf{val}}{\Gamma \vdash \textit{TopLevelObject} : (\Gamma_1 \cup \{\text{‘}\alpha\text{’} : \mathrm{ran}(\Gamma_1)\})\mathsf{val}} \tag{6.108}$$

[combine BindSpec]

$$\frac{\Gamma \vdash \textit{BindSpec}_1 : \Gamma_1\mathsf{val} \quad \Gamma \vdash \textit{BindSpec}_2 : \Gamma_2\mathsf{val} \quad \mathrm{dom}(\Gamma_1) \cap_\sigma \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash \textit{BindSpec}_1\textit{BindSpec}_2 : (\Gamma_1 \cup_\sigma \Gamma_3)\mathsf{val}} \tag{6.109}$$

with

$$\Gamma_3 = \Gamma_2 \setminus \{\text{‘}\alpha\text{’} : \Gamma_2(\text{‘}\alpha\text{’})\} \tag{6.110}$$

6.8.3. Objects, Classes and Types

The binding of objects and types, the main structures in the binding, consists primarily of assignments defined by the *VAssign* production explained later in sec. 6.8.4 and secondarily of structures defining the target of the assignments. These structures recall the class definitions and are ordered in a hierarchy. The binding of classes, only allowed in the binding of the prelude, is similar to those of objects, but the object identifier is missing. The binding is based on following grammar:

$inType$ ::= ‘in’ ‘type’ $\boldsymbol{x}$ $VAssign$ ‘end’ ‘;’

$inClass$ ::= ‘class’ $\boldsymbol{x}$ $OAssign$ ‘end’ ‘;’

$inObject$ ::= ‘object’ $\boldsymbol{x}_1$ ‘:’ $\boldsymbol{x}_2$ $OAssign$ ‘end’ ‘;’

$inSelector$ ::= ‘in’ $\boldsymbol{x}$ ‘:’ ‘Selector’ $Path$ ‘end’ ‘;’

$inLoop$ ::= ‘in’ $\boldsymbol{x}$ ‘:’ ‘Loop’ $OAssign$ ‘end’ ‘;’

$Path$ ::= $Path_1$ $Path_2$ | ‘on’ $OAssign$ ‘end’ ‘;’

$OAssign$ ::= $RebindTypes$ $BindObjects$ $ValueAssign$ $inSuper$

$ValueAssign$ ::= | $VAssign$

$RebindTypes$ ::= | $inType$ $RebindTypes$

$BindObjects$ ::= | $BindObjects_1$ $BindObjects_2$ | $inObject$ | $inLoop$ | $inSelector$

$inSuper$ ::= | 'in' 'super' $VAssign$ $inSuper$ 'end'

The typing of assignments of fuzzy type elements requires that the type is defined in the current environment. The l-values are the type's terms and modifiers. Note that the reserved word 'universe' and its type attribute remain in the l-value set. This it is not harmful: 'universe' cannot be used as an ordinary identifier in the specification, hence no clashes with other identifiers can occur.

$$\text{[in type]}\quad \frac{\Gamma \vdash x : \Gamma_1 \mathsf{Ftype} \quad \Gamma \sqcup (\text{`}\lambda\text{'} : \Gamma_1) \vdash \mathit{VAssign} : \Gamma_2 \mathsf{val}}{\Gamma \vdash (\text{`in' `type'}\ x\ \mathit{VAssign}\ \text{`end' `;'}) : \{x : \Gamma_2 \mathsf{val}\} \mathsf{val}} \tag{6.111}$$

The assignment of modifiers and terms in handled later from rule (6.130) onwards together with assignments in classes.

The *inObject*-production is handled in rules (6.112) and (6.115). L-values of classes are all exported and private elements; inherited private elements are considered later in rule (6.129). Rule (6.112) handles assignments of top level objects and requires only that they conform to class Scene. For assignments inside the current object the rule introduces the proper l-values in (6.113), where alpha and omega are not allowed as l-values.

$$\text{[in object initial]}\quad \frac{\begin{array}{c}\Gamma \vdash x_2 : \Gamma_1 \mathsf{Class} \qquad x_1 \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \lambda : \emptyset \\ \Gamma \sqcup \Gamma_2 \vdash \mathit{OAssign} : \Gamma_3 \mathsf{val} \quad \Gamma_1 \mathsf{Class} <: \Gamma(\text{`Scene'})\end{array}}{\Gamma \vdash (\text{`object'}\ x_1\ \text{`:'}\ x_2\ \mathit{OAssign}\ \text{`end' `;'}) : \{x_1 : \Gamma_3 \mathsf{val}, \text{`}\sigma\text{'} : \{x_1 : \Gamma_1 \mathsf{Class}\}\} \mathsf{val}} \tag{6.112}$$

where

$$\Gamma_2 = \{x_1 : \Gamma_1 \mathsf{class}\} \sqcup \Gamma_1 \sqcup \{\text{`}\lambda\text{'} : (\Gamma_1(\text{`exports'}) \cup \Gamma_1(\text{`let'})) \setminus \Gamma_4\} \tag{6.113}$$

$$\Gamma_4 = \{\text{`alpha'} : \Gamma_2(\boldsymbol{exports})(\text{`alpha'}), \text{`omega'} : \Gamma_2(\boldsymbol{exports})(\text{`omega'})\} \tag{6.114}$$

Rule (6.115) is used for all other objects in the hierarchy of objects and is only applicable if the current object x_1 is proper l-value, i.e. it is a member of $\Gamma(\lambda)$. However, as assignments to alpha and omega are not allowed, we check explicitly that x_1 is unequal to these identifiers. It is required that the type of x_2 is a proper subtype of the type of x_1 to fulfill the laws of substitutability (cf. sec. 5.3.2.1 on page 65). Again we remove alpha and omega from the set of l-values.

$$\text{[in object]}\quad \frac{\begin{array}{c}\Gamma(\text{`}\lambda\text{'}) \vdash x_1 : \Gamma_1 \mathsf{class} \quad x_1 \notin \{\text{`alpha'}, \text{`omega'}\} \quad \Gamma \vdash x_2 : \Gamma_2 \mathsf{Class} \\ \Gamma_2 \mathsf{Class} <: \Gamma_1 \mathsf{Class} \quad \Gamma \sqcup \Gamma_3 \vdash \mathit{OAssign} : \Gamma_3 \mathsf{val}\end{array}}{\Gamma \vdash (\text{`object'}\ x_1\ \text{`:'}\ x_2\ \mathit{OAssign}\ \text{`end' `;'}) : \{x_1 : \Gamma_3 \mathsf{val}\} \mathsf{val}} \tag{6.115}$$

where

$$\Gamma_3 = \Gamma_2 \sqcup \{\text{`}\lambda\text{'} : (\Gamma_1(\text{`exports'}) \cup \Gamma_1(\text{`let'})) \setminus \Gamma_4\} \tag{6.116}$$

$$\Gamma_4 = \{\text{`alpha'} : \Gamma_2(\textbf{\textit{exports}})(\text{`alpha'}), \text{`omega'} : \Gamma_2(\textbf{\textit{exports}})(\text{`omega'})\} \tag{6.117}$$

The rule for classes (6.118) mimics exactly those for objects (6.112), since class bindings occur also on the top-level. However, we omit the requirement that the bound class inherits from Scene and we use the name of the class also as the identifier of the value binding. This allows simple checking that no class is bound twice in the prelude's binding, by requiring that the class identifier does not occur in the domain of Γ. In contrast to the object bindings above, we do not exclude alpha and omega from the l-value set, if the class bound is class Interval: we need to set alpha and omega once.

[in class]

$$\frac{\Gamma \vdash \boldsymbol{x} : \Gamma_1\mathsf{Class} \quad \boldsymbol{x} \notin \mathrm{dom}(\Gamma) \quad \Gamma \sqcup \Gamma_2 \vdash \textbf{\textit{OAssign}} : \Gamma_3\mathsf{val}}{\Gamma \vdash (\text{`class'}\ \boldsymbol{x}\ \textbf{\textit{OAssign}}\text{`end' `;'}) : \{\boldsymbol{x} : \Gamma_2\mathsf{val}\}} \tag{6.118}$$

where

$$\Gamma_2 = \{\boldsymbol{x} : \Gamma_1\mathsf{class}\} \sqcup \Gamma_1 \sqcup \{\text{`}\lambda\text{'} : (\Gamma_1(\text{`exports'}) \cup \Gamma_1(\text{`let'}))\} \setminus \Gamma_4 \tag{6.119}$$

$$\Gamma_4 = \begin{cases} \emptyset & \boldsymbol{x} = \text{`Interval'} \\ \{\text{`alpha'} : \Gamma_2(\textbf{\textit{exports}})(\text{`alpha'}), \text{`omega'} : \Gamma_2(\textbf{\textit{exports}})(\text{`omega'})\} & \text{otherwise} \end{cases} \tag{6.120}$$

Loops (6.121) and selectors (6.124) share a similar value structure, because each local declaration and rule block (i.e. the loop body and the alternative paths) has its own type environment using identifier on to indicate that we do not have an ordinary object. For selectors each path has to be assigned in the same order as the paths are declared (6.123).

[in loop]

$$\frac{\Gamma(\text{`}\lambda\text{'}) \vdash \boldsymbol{x} : \Gamma_1\mathsf{Loop} \quad \Gamma \sqcup \Gamma_1(\text{`on'}) \vdash \textbf{\textit{OAssign}} : \Gamma_3\mathsf{val}}{\Gamma \vdash (\text{`in'}\ \boldsymbol{x}\ \text{`:' `Loop'}\ \textbf{\textit{OAssign}}\ \text{`end' `;'}) : \{\boldsymbol{x} : \{\text{`on'} : \Gamma_3\mathsf{val}\}\mathsf{val}\}\mathsf{val}} \tag{6.121}$$

[Path]

$$\frac{\Gamma \vdash \textbf{\textit{OAssign}} : \Gamma_1\mathsf{val}}{\Gamma \vdash \text{`on'}\ \textbf{\textit{OAssign}}\ \text{`end'`;'} : \Gamma_1\mathsf{val}} \tag{6.122}$$

[Combine Paths]

$$\frac{\Gamma \vdash \textbf{\textit{Path}}_1 : \Gamma_1\mathsf{val} \quad \Gamma \sqcup \Gamma_1(\text{`on'}) \vdash \textbf{\textit{Path}}_2 : \Gamma_3\mathsf{val}}{\Gamma \vdash \textbf{\textit{Path}}_1\ \textbf{\textit{Path}}_2 : (\Gamma_1 \cup \{\text{`on'} : \Gamma_3\mathsf{val}\}\mathsf{val}} \tag{6.123}$$

[in selector]

$$\frac{\Gamma(\text{`}\lambda\text{'}) \vdash \boldsymbol{x} : \Gamma_1\mathsf{Sel} \quad \Gamma \sqcup \Gamma_1(\text{`on'}) \vdash \textbf{\textit{Path}} : \Gamma_3\mathsf{val}}{\Gamma \vdash (\text{`in'}\ \boldsymbol{x}\ \text{`:' `Selector'}\ \textbf{\textit{Path}}\ \text{`end' `;'}) : \{\boldsymbol{x} : \{\text{`on'} : \Gamma_3\mathsf{val}\}\mathsf{val}\}\mathsf{val}} \tag{6.124}$$

The binding of single objects is handled in rule (6.125) also dealing with contexts which re-bind existing type definitions. Its type attribute Γ_5 is the union of its sub-terms attributes, but omitting re-bound fuzzy types (6.126). Assignments to private inherited elements are handled as in rule (6.129).

$$[\text{object assignment}]\quad \frac{\Gamma \vdash \textit{RebindTypes} : \Gamma_1\mathsf{val} \quad \Gamma \vdash \textit{BindObjects} : \Gamma_2\mathsf{val} \quad \Gamma \vdash \textit{VAssign} : \Gamma_3\mathsf{val} \quad \Gamma \vdash \textit{inSuper} : \Gamma_4\mathsf{val}}{\Gamma \vdash (\textit{RebindTypes BindObjects VAssign inSuper}) : \Gamma_5\mathsf{val}} \tag{6.125}$$

where

$$\Gamma_5 = (\Gamma_2 \cup \Gamma_3 \cup \{\text{`let'} : \Gamma_4\mathsf{val}\})\mathsf{val} \tag{6.126}$$

Rule (6.125) applies recursively rules (6.111) and (6.115) in productions *RebindTypes* and *BindObjects*, resp., with corresponding type rules (6.127) and (6.128). Private inherited elements are encoded in the type attribute of the special element ‘let’. In rule (6.129) we deal with assignments to private elements and thus replace the current l-values with these elements. We do this again recursively for the next nested ‘let’-block while walking up the inheritance relation with the *inSuper* production.

$$[\text{re-bind type}]\quad \frac{\Gamma \vdash \textit{inType} : \Gamma_1\mathsf{val} \quad \Gamma \vdash \textit{RebindTypes} : \Gamma_2\mathsf{val} \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash (\textit{inType RebindTypes}) : (\Gamma_1 \cup \Gamma_2)\mathsf{val}} \tag{6.127}$$

$$[\text{bind objects}]\quad \frac{\Gamma \vdash \textit{BindObjects}_1 : \Gamma_1\mathsf{val} \quad \Gamma \vdash \textit{BindObjects}_2 : \Gamma_2\mathsf{val} \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash (\textit{BindObjects}_1\ \textit{BindObjects}_2) : (\Gamma_1 \cup \Gamma_2)\mathsf{val}} \tag{6.128}$$

$$[\text{super}]\quad \frac{\Gamma \sqcup (\text{`}\lambda\text{'} : \Gamma(\text{`}\lambda\text{'})(\text{`let'})) \vdash \textit{VAssign} : \Gamma_1\mathsf{val} \quad \Gamma \sqcup (\text{`}\lambda\text{'} : \Gamma(\text{`}\lambda\text{'})(\text{`let'})) \vdash \textit{inSuper} : \Gamma_2\mathsf{val}}{\Gamma \vdash (\text{`in' `super'}\ \textit{VAssign inSuper}\ \text{`end'`;'}) : (\Gamma_1 \cup \{\text{`let'} : \Gamma_2\mathsf{val}\})\mathsf{val}} \tag{6.129}$$

6.8.4. Assignment of Values

Assignment of values to identifiers demands that the identifiers are l-values. We assign three kinds of values: modifiers, fuzzy sets and references. Note that objects are handled by the *inObject* production and thus objects are not considered as values for assignments. Fuzzy set values may be either constants defined by the various fuzzy set construction mechanism or they refer to terms of fuzzy types. The grammar of

assignments differentiates between identifers, fuzzy set expressions and references to top level objects by ref. We discuss these kinds of assignments in this section.

$VAssign$ ::= 'let' $\boldsymbol{x}$ ':=' $Expr$ ';' | $VAssign$ $VAssign$

$Expr$ ::= I | $FSExpr$ | 'ref' '(' x ')'

All assignments and their combination have as type attribute a value environment. The rules (6.130), (6.131) and (6.132) require that the declared type of x and the type of the expression are compatible. The universe U of the declared type of x is added to the type environment for checking if the expression matches the type of x. The latter two rules (6.131) and (6.132) handle terms and modifiers in fuzzy types. The former rule (6.130) handles assignments of fuzzy sets to attributes of objects. Because of type F for *Expr*, assignments to x with values of other attributes are impossible, because these values would have type $\Gamma\mathsf{ftype}$. Hence only assignments to x with constants or terms of fuzzy types are possible. In (6.133) references are assigned. We check that the declared type of x_1 conforms to class Scene. The type attribute of this assignment is a references environment $\{x_2 : \Gamma_1\}\mathsf{ref}$ embedded in a value environment for x_1. Finally in (6.134) we check that no multiple assignments to variables are possible.

[assign fuzzy set]

$$\frac{\Gamma(\text{'}\lambda\text{'}) \vdash \boldsymbol{x} : \Gamma_1\mathsf{ftype} \quad \Gamma_1 \vdash \text{'universe'} : F \quad \Gamma \vdash \textit{Expr} : F}{\Gamma \vdash (\text{'let' } \boldsymbol{x} \text{ ':=' } \textit{Expr} \text{ ';'}) : \{\boldsymbol{x} : \Gamma_1\}\mathsf{val}} \tag{6.130}$$

[assign term]

$$\frac{\Gamma(\text{'}\lambda\text{'}) \vdash \boldsymbol{x} : F \quad \Gamma(\text{'}\lambda\text{'}) \vdash \text{'universe'} : F \quad \Gamma \vdash \textit{Expr} : F}{\Gamma \vdash (\text{'let' } \boldsymbol{x} \text{ ':=' } \textit{Expr} \text{ ';'}) : \{\boldsymbol{x} : \Gamma_1\}\mathsf{val}} \tag{6.131}$$

[assign modifier]

$$\frac{\Gamma(\text{'}\lambda\text{'}) \vdash \boldsymbol{x} : F \to F \quad \Gamma(\text{'}\lambda\text{'}) \vdash \text{'universe'} : F \quad \Gamma \vdash \textit{Expr} : F \to F}{\Gamma \vdash (\text{'let' } \boldsymbol{x} \text{ ':=' } \textit{Expr} \text{ ';'}) : \{\boldsymbol{x} : F \to F\}\mathsf{val}} \tag{6.132}$$

[assign ref]

$$\frac{\Gamma(\text{'}\lambda\text{'}) \vdash x_1 : \Gamma_1\mathsf{class} \quad \Gamma \vdash \text{'Scene'} : \Gamma_2\mathsf{Class} \quad \Gamma_1\mathsf{Class} <: \Gamma_2\mathsf{Class}}{\Gamma \vdash (\text{'let' } x_1 \text{ ':=' 'ref' '(' } x_2 \text{ ')'}) : \{x_1 : \{x_2 : \Gamma_1\}\mathsf{ref}\}\mathsf{val}} \tag{6.133}$$

[Combine value assignments]

$$\frac{\Gamma \vdash \textit{VAssign}_1 : \Gamma_1\mathsf{val} \quad \Gamma \vdash \textit{VAssign}_2 : \Gamma_2\mathsf{val} \quad \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset}{\Gamma \vdash \textit{VAssign}_1\, \textit{VAssign}_2 : (\Gamma_1 \cup \Gamma_2)\mathsf{val}} \tag{6.134}$$

Fuzzy set constants can either be enumerated sets, built with the convex or the generic constructor. The *convex* production denotes the convex constructors as defined in production *ConvexConstruction* in sec. A.1.7.4 on page 272.

$FSExpr$ ::= $FSEnum$ | $FSConvex$ | $FSGeneric$

$FSEnum$ ::= ‘{’ $VList$ ‘}’

$VList$ ::= ‘(’ $\boldsymbol{x}$ ‘,’ $\boldsymbol{n}$ ‘)’ | $VList_1$ ‘,’ $VList_2$

$FSConvex$ ::= $convex$ ‘(’ $AList$ ‘)’

$AList$ ::= $\boldsymbol{n}$ | $AList_1$ ‘,’ $AList_2$

$FSGeneric$ ::= ‘linear’ ‘(’ $MList$ ‘)’

$MList$ ::= ‘(’ $\boldsymbol{n}$ ‘,’ $\boldsymbol{m}$ ‘)’ | $MList_1$ ‘,’ $MList_2$

Enumerated fuzzy sets are a list of pairs consisting of identifiers and membership values. The universe of enumerated fuzzy sets is a type environment containing the set of identifiers with their type attribute u. In rule (6.135) we assign the type attribute $\Gamma \rightarrow$ f to each pair of identifiers and membership values, where Γ here is the environment Γ_1 used in rule (6.137) to type check *VList*. The type attribute is propagated to rule (6.136) and results in the final type of the fuzzy set expression in rule (6.137).

$$[\text{vlist}] \quad \frac{\Gamma \vdash \boldsymbol{x} : \mathsf{u} \quad \boldsymbol{n} : \mathsf{num} \quad 0.0 \leq \boldsymbol{n} \leq 1.0}{\Gamma \vdash \text{‘(’ } \boldsymbol{x} \text{ ‘,’ } \boldsymbol{n} \text{ ‘)’} : (\Gamma \rightarrow \mathsf{f})} \tag{6.135}$$

$$[\text{combine vlist}] \quad \frac{\Gamma \vdash \boldsymbol{VList_1} : \boldsymbol{F} \quad \Gamma \vdash \boldsymbol{VList_2} : \boldsymbol{F}}{\Gamma \vdash (\boldsymbol{VList_1} \text{ ‘,’ } \boldsymbol{VList_2}) : \boldsymbol{F}} \tag{6.136}$$

$$[\text{enumerated fuzzy set}] \quad \frac{\Gamma \vdash \text{‘universe’} : \Gamma_1 \rightarrow \mathsf{f} \quad \Gamma_1 \vdash \boldsymbol{VList} : \boldsymbol{F}}{\Gamma \vdash (\text{‘\{’ } \boldsymbol{VList} \text{ ‘\}’}) : \boldsymbol{F}} \tag{6.137}$$

Fuzzy sets created by a convex fuzzy constructor require that the universe is numeric and that arguments of the constructor are also numeric. We use the type num also for the list of parameters. We do not check the bounds of the arguments because they depend on the parameterization of constructors.

$$[\text{combine argument list}] \quad \frac{\Gamma \vdash \boldsymbol{AList_1} : \mathsf{num} \quad \Gamma \vdash \boldsymbol{AList_2} : \mathsf{num}}{\Gamma \vdash (\boldsymbol{AList_1} \text{ ‘,’ } \boldsymbol{AList_2}) : \mathsf{num}} \tag{6.138}$$

$$[\text{convex fuzzy set}] \quad \frac{\Gamma \vdash \text{‘universe’} : [\boldsymbol{n}, \boldsymbol{m}] \rightarrow \mathsf{f} \quad \Gamma \vdash \boldsymbol{AList} : \mathsf{num}}{\Gamma \vdash (\boldsymbol{convex} \text{ ‘(’ } \boldsymbol{AList} \text{ ‘)’ }) : ([\boldsymbol{n}, \boldsymbol{m}] \rightarrow \mathsf{f})} \tag{6.139}$$

The linear constructor requires as arguments pairs of numeric elements and their membership values, similar to enumerated fuzzy sets. Thus, we mimic somewhat the

rules mentioned above. In rule (6.140) each element needs to be an element of the universe, i.e. inside the bounds of the universe. The type of each pair is that of a fuzzy type with a singleton universe. In rule (6.141) we concatenate two member lists. The order of the member lists is relevant, because the arguments (more precisely: the first element of each pair) of the linear constructor are required to be strictly ascending, which checked by $n_2 < n_3$. As result we get as universe the smallest interval containing both member lists' universes. Finally, in rule (6.142), we require that the expression's universe ($[n, m]$) contains the universe of the arguments of 'linear' ($[n_1, m_1]$).

$$\text{[member list element]} \quad \frac{\Gamma \vdash \text{`universe'} : [n, m] \to \mathsf{f} \quad \Gamma \vdash n_1 : \mathsf{num} \quad n \leq n_1 \leq m \quad \Gamma \vdash n_2 : \mathsf{num} \quad 0.0 \leq n_2 \leq 1.0}{\Gamma \vdash (\text{ `(' } n_1 \text{ `,' } n_2 \text{ `)' }) : ([n_1, n_1] \to \mathsf{f})} \tag{6.140}$$

$$\text{[combine member list]} \quad \frac{\Gamma \vdash \mathit{MList}_1 : ([n_1, n_2] \to \mathsf{f}) \quad \Gamma \vdash \mathit{MList}_2 : ([n_3, n_4] \to \mathsf{f}) \quad n_2 < n_3}{\Gamma \vdash (\mathit{MList}_1 \text{ `,' } \mathit{MList}_2) : ([n_1, n_4] \to \mathsf{f})} \tag{6.141}$$

$$\text{[generic fuzzy set]} \quad \frac{\Gamma \vdash \text{`universe'} : [n, m] \to \mathsf{f} \quad \Gamma \vdash \mathit{MList} : [n_1, m_1] \to \mathsf{f} \quad n \leq n_1 \quad m_1 \leq m}{\Gamma \vdash (\text{`linear' `(' } \mathit{MList} \text{ `)' }) : ([n, m] \to \mathsf{f})} \tag{6.142}$$

6.9. Remarks

We finish this chapter with some remarks concerning our approach of defining the static semantics of Vitruv$_L$.

Often the static semantics is heavily intertwined with the evaluation semantics. We separate both because of the complexity we would gain by discussing them together. This is the reason for not considering semantics of types, because this influenced immediately by the evaluation semantics.

We do not prove soundness and completeness of our type system, because more than one hundred axioms and inference rules are to much to be proven by hand, an automatic theorem prover would be required to do the structural induction on the term construction. In the (research) literature we often find approaches introducing new primitives into existing caluculi, such as introducing sub-typing to the typed lambda calculus: in this case we need certainly a thorough analysis. In contrast to this situation, we do not introduce any new type system primitives to the known systems in the literature, but merely apply well known techniques. Therefore, we think it is regretful to omit the proofs in our case.

Following Cardelli's approach, we would model classes as recursive types consisting of a union of the unit type (denoting nil-values) and a record-like structure. We do not follow this path of explicit recursive types but apply the approach of Schmidt

(1994). Nevertheless, it is important to notice that our definitions introduce implicitly a recursive type in the type attribution of classes. In the binding we have the corresponding unfolding, where we use the in-expression to look in the next unfolding of the class definition. Also in the binding, we notate in the value type environments, $\Gamma\mathsf{val}$ all elements with values, i.e. all those variables $x : \theta \in \Gamma\mathsf{Class}$, which are not in the corresponding $\Gamma\mathsf{val}$, have no value. This finishes recursion and correlates to the one and only value of the unit type as used by Cardelli in his example for lists. In this respect, our approach uses the same concepts and our approach should be sound, too.

7. Vitruv_I, the Intermediate Language for Vitruv_L

In chapter 6 we discussed the static semantics of Vitruv_L as a formal type system. Now we move to the dynamic semantics or the behavior of Vitruv_L. We differentiate between event-free and event-based behaviors of presentations specified with Vitruv_L. In this chapter we focus on event-free behaviors, in chapter 8 on page 149 we consider also event-based behaviors.

In this chapter we start with characterizing the event-free behavior and discussing why we need the language Vitruv_I. This is followed by the language description of Vitruv_I and the presentation of its operational semantics. After that we show how to transform a Vitruv_L specification into Vitruv_I.

7.1. Event-Free Behavior

The event-free behavior of Vitruv_L contains all interval activations without events and event reactions, i.e. without events, loops and selectors. Following the discussion concerning loop and selector bodies in sec. 5.5.1 on page 70, selector and loop bodies are independent of their surroundings. Therefore, we can partition the event-free behavior of Vitruv_L such that loop and selector bodies are separated from each other and from class bodies. We call these partitions ***blocks***. In sec. 7.4.5 on page 146 we present the details of how blocks are determined from a Vitruv_L specification.

The most interesting characteristic of event-free behavior is that its features can be checked statically, i.e. without executing or simulating the specified (part of the) presentation. The most important question is whether the specified behavior is consistent, i.e. whether it can really happen. The constraint solving algorithm by Allen deals with this question. This algorithm, however, requires as input the set of all intervals and their specified relations of a block as a flat structure. Therefore, we cannot use a Vitruv_L specification immediately, but we have to linearize Vitruv_L specifications, i.e. we have to resolve inheritance structures, compound relationships, constraints, interval instantiation and nested assignments in the binding.

To ease the linearization process, we introduce Vitruv_I, a simple imperative language, which is the target language of the linearization. Within Vitruv_I we have a flat name-space with intervals and relations, blocks, and fuzzy values with constraints

and assignments. The execution of a $Vitruv_l$ program results in a state structure consisting of blocks, intervals, relations and durations. We use this structure as input for consistency checking algorithms such as Allen's constraint solver or the more advanced algorithm by Christoph Begall (2002).

The results of executing $Vitruv_l$ programs is also needed for modeling the behavior of $Vitruv_L$ specifications with Vitruvian Nets (sec. 8 on page 149). An important observation is that if we properly glue models for the blocks together with models for the respective events and event reactions, we can reconstruct the entire behavior of $Vitruv_L$ specifications. This is the main idea for deriving Vitruvian Nets from $Vitruv_L$ specifications. As for the constraint solving, we require again the flat structure of the event-free behavior for constructing FTVNs (sec. 8.2 on page 153), which are major building blocks for the construction of Vitruvian Nets. The detailed composition of event-free and event-based behaviors is presented in sec. 8.6 on page 183.

7.2. Language Description

The objective of the intermediate language $Vitruv_l$ is to define a complete $Vitruv_L$ specification in terms of objects excluding events and event reactions. $Vitruv_l$ is an imperative language the statements of which modify the complex state of the system. The state is a set of finite sets, functions and relations, capturing the deterministic structure of $Vitruv_L$. That means that we track elements of objects, in particular the length-element, and scenes, loop and selector bodies, which we subsume as ***block*** in $Vitruv_l$. The state of $Vitruv_l$ is the basis for consistency checking of a $Vitruv_L$ specification, as mentioned above.

block

name	*meaning*
interval (x)	declare x as an interval
fuzzyvalue (x)	declare x as a fuzzy value
modifier (x)	declare x as a modifier
block (x)	declare x as a block
element (x,y)	x is an element of interval (class) y
inBlock (x,y)	interval x is in block y
constraint (x,f)	fuzzy value x has to fulfill constraint f
length (x,y)	y is the length element of x
value (x)	yields the fuzzy value of x
update (x, f)	updates the fuzzy value of x by f
holds (x, y, r1, . . . , rn)	one of relations r1, . . . , rn holds between x and y

Table 7.1.: Statements in $Vitruv_l$

In table 7.1 on the preceding page we present the statements of Vitruv_I, where identifiers are denoted with x and y. Fuzzy set expressions are denoted with f and they are constructed following productions *ConvexConstruction* and *GenericConstruction* as defined for the binding of Vitruv_L (sec. A.1.7.4 on page 272). We use inf and -inf to denote ∞ and $-\infty$, resp., used as parameters for the construction of unbounded fuzzy sets. Interval relations, used as arguments of the holds statement, are encoded versions of Allen's thirteen relations using the first letter except for = (equals), mi (metby), fi (finishedby), si (startedby) and oi (overlappedby).

The four statements interval, fuzzyvalue, modifier and block declare new intervals, fuzzy values, modifiers and blocks, respectively. In Vitruv_I, we apply declaration before use. Fuzzy values and modifiers can be updated by update and additionally a constraint for fuzzy values can be defined by constraint. Values and constraints have to match: it is only allowed to update a fuzzy value if the new value satisfies the constraint. This can be checked statically. The dependency between value and constraint is independent of their order of assignment: the value has always to be a subset of the constraint, the lack of a constraint means every value is allowed, hence every value will match. For uniform treatment, all fuzzy sets are elements of $\mathcal{F}(\mathbb{R})$, i.e. we encode integer based and enumerated sets into universe $\mathbb{R}$.

Elements of classes, which are instances of fuzzy types, and in particular element length, are defined for each interval by element and length. It is required that the elements are already declared as fuzzy values. Similarly, the relationship between intervals and their block is defined by inBlock, the interval and the block have to be declared. It is sufficient to define this relationship only for those intervals that are explicitly declared in the block, i.e. only those declared in the let-part of a loop or a conditional body, or as elements of a scene. The entire set of intervals inside a block can easily calculated by interval relations.

Finally, we have the holds statement in Vitruv_I defining interval relationships. Each holds statement defines a disjunction of possible relations between its two intervals. If two or more holds statements for the same two intervals are used, then the resulting interval relation is the intersection of all these relation sets, because – as usual in Allen's calculus – we have conjunctions of disjunctive relations.

In spec. 7.1 on the next page we show a simple example of a Vitruv_I specification. It is an excerpt of the binding of class Interval, where we introduce fuzzy value zero (from fuzzy type DURATION) and setting its value to singleton(0.0). It is followed by defining intervals alpha, omega and this and their length elements. After that we establish the relations between this, alpha and omega, finishing with the length constraints that alpha and omega have length zero and that length may have an arbitrary (positive) duration.

Later, in sec. 7.4 on page 142, we present more examples while showing how to transform Vitruv_L specifications into Vitruv_I specifications.

Specification 7.1 A simple Vitruv$_I$ example

```
fuzzyvalue (zero).
update (zero, singleton (0.0)).
interval (alpha).
interval (omega).
interval (this).
fuzzyvalue (length).
fuzzyvalue (alphalength).
fuzzyvalue (omegalength).
element (length, this).
length (length, this).
element (alphalength, alpha).
length (alphalength, alpha).
element (omegalength, alpha).
length (omegalength, alpha).
holds (alpha, this, { s }).
holds (omega, this, { f }).
holds (alpha, omega, {b, m}).
constraint (alphalength, value (zero)).
constraint (omegalength, value (zero)).
update (length, rectangle (0.0, inf)).
```

7.3. Semantics of Vitruv_l

The semantics of Vitruv_l define how a set of terms is executed and transformed into an internal representation, which is the basis of any further analysis, in particular for checking the consistency of the specification.

7.3.1. Abstract Syntax of Vitruv_l

The abstract syntax of Vitruv_l uses again identifiers from an infinite set *Id* with *i* ranging through *Id*, *f* is a fuzzy set literal. In contrast to the concrete grammar used in examples of Vitruv_l, we omit parentheses around predicate arguments in terms and parentheses for structuring expressions because of simplicity reasons.

$statement$ ::= $term$ '.' | $statement$ $statement$

$term$ ::= 'interval' i | 'element' i ',' i
| 'fuzzyvalue' i | 'modifier' i | 'block' i | 'inBlock' i ',' i
| 'constraint' i ',' e | 'length' i ',' i
| 'holds' i ',' i ',' rel
| 'update' i ',' e

e ::= f | 'value' i | 'apply' i e | $defmod$ | e $binop$ e

$defmod$ ::= 'not' | 'norm' | 'very' | 'mol' | 'somewhat'
| 'plus' | 'extremely' | 'intensify' | 'slightly' | 'above' | 'below'

$binop$ ::= 'or' | 'and'

rel ::= '{' rs '}'

rs ::= rs ',' rs | r

r ::= '=' | 'm' | 'mi' | 'f' | 'fi' | 's' | 'si' | 'a' | 'b' | 'd' | 'c' | 'o' | 'oi'

7.3.2. Semantical Objects

The semantical structure of Vitruv_l consists of various maps between references and values, and between references themselves denoting the interval structures.

We use the following notation for finite functions g, g_1 and g_2:

$$Dom(g) \quad \text{domain of } g \tag{7.1}$$

$$Ran(g) \quad \text{range of } g \tag{7.2}$$

$$\{\} = \emptyset \quad \text{the empty function} \tag{7.3}$$

$$g(x_1) = x_2 \Leftrightarrow (x_1 \mapsto x_2) \in g \tag{7.4}$$
$$g_1 \oplus g_2 \quad \text{functional override with:} \tag{7.5}$$
$$(g_1 \oplus g_2)(x_1) = \begin{cases} g_2(x_1) & \text{if } x_1 \in \mathit{Dom}(g_2) \\ g_1(x_1) & \text{otherwise} \end{cases}$$

Basic semantical objects are a set of assigned references, $X \subset \mathit{Ref}$, a set of fuzzy sets, $F \subset \mathcal{F}(\mathbb{R})$, the set M of the eleven modifiers, and Allen's 13 interval relations, R. We use x, f, m, r and p to range through X, F, M, R and $\mathcal{P}(R)$, respectively. We assume that Ref is disjoint to R, M and F.

Additionally we have the (finite) sets and functions denoting the global state of Vitruv$_I$ shown in tab. 7.2, their meaning is similar to the statements of Vitruv$_I$. Sets ***interval***, ***fuzzyvalue***, ***modifier*** and ***block*** have as elements those references denoting intervals, fuzzy values, modifiers and blocks, respectively. The following functions map references: ***element*** maps fuzzy values to their respective intervals, ***inBlock*** maps intervals to their block, and ***length*** maps intervals to their length element. The next two functions map references to fuzzy values and modifiers to their fuzzy set and modifier values: ***constraint*** maps fuzzy values to their constraint fuzzy sets, and ***value*** maps fuzzy values and modifiers to their fuzzy set and modifier values, respectively. Finally, function ***holds*** maps references, (x_1, x_2), of a pair of intervals to the set of interval relations between x_1 and x_2.

Name	*Signature*
interval	$\subset X$
fuzzyvalue	$\subset X$
modifier	$\subset X$
block	$\subset X$
element	$\in \mathit{fuzzyvalue} \rightarrow \mathit{interval}$
inBlock	$\in \mathit{interval} \rightarrow \mathit{block}$
constraint	$\in \mathit{fuzzyvalue} \rightarrow F$
length	$\in \mathit{interval} \rightarrow \mathit{fuzzyvalue}$
value	$\in \mathit{fuzzyvalue} \cup \mathit{modifier} \rightarrow F \cup M$
holds	$\in \mathit{interval} \times \mathit{interval} \rightarrow \mathcal{P}(R)$

Table 7.2.: Sets and functions for the global state of Vitruv$_I$.

A particular state, S, of Vitruv$_I$ is a labeled 11-tuple of base set X and those ten named sets and functions defined in the tab. 7.2. We access each component of S by its name:

$$S = (X, \mathit{interval}, \mathit{fuzzyvalue}, \mathit{modifier}, \mathit{block}, \mathit{element}, \tag{7.6}$$

$$inBlock, constraint, length, value, holds)$$

Semantical objects M and R are omitted, because they are constant. The set of used fuzzy sets, F, is omitted for simplicity of the notation: the particular set of fuzzy sets is not of interest and as reservoir of values we simply use $\mathcal{F}(\mathbb{R})$, which is again constant and thus is omitted.

Updates of only one component of state S is denoted for simplicity as a form of a functional override. In (7.7), S' is equal to S except that component *interval* has the additional element x (assuming $x \notin S$), in (7.8), S'' differs from S only in component *element*.

$$\begin{aligned}
S' &= S \oplus (S.interval \cup \{x\}) && (7.7)\\
&= (S.X, S.interval \cup \{x\}, S.fuzzyvalue, \ldots, S.holds)\\
S'' &= S \oplus (S.element \oplus \{x_1 \mapsto x_2\}) && (7.8)\\
&= (S.X, S.interval, \ldots, S.block, S.element \oplus \{x_1 \mapsto x_2\}, S.inBlock, \ldots, S.holds)
\end{aligned}$$

7.3.3. Operational Semantics

Before defining the operational semantics of Vitruv$_l$, we need some additional structures and functions.

For bookkeeping reasons we use an environment E mapping identifers to references:

$$E : Id \to X \tag{7.9}$$

Function *allocate* allocates a fresh reference $x \in Ref$ and extends the set of allocated references X. Function *new* introduces a new identifier, i, and updates accordingly state, S, and environment, E:

$$\begin{aligned}
allocate(X) &:= (x, X \cup \{x\}) \quad \text{where } x \notin X && (7.10)\\
new(i, E, S) &:= (E', S') \quad \text{with} && (7.11)\\
(x, X') &= allocate(S.X)\\
E' &= E \oplus \{(i \mapsto x)\}\\
S' &= S \oplus X'
\end{aligned}$$

We define the semantics of Vitruv$_l$ as a structural operational semantics (Winskel, 1993; Mitchel, 1996). It is a deduction system similar to the static semantics, consisting of axioms and rules. The judgments used have the form

$$E, S \vdash t \models \langle E', S', v\rangle$$

meaning that the evaluation of (compound) term t in state S and with environment E results in a new state S', a new environment E' and a value $v \in F \cup M \cup \mathcal{P}(R) \cup \{ok\}$.

If we have expressions, then v will hold the respective values and the value for *ok* is discarded, if we have commands, then v is *ok* if the execution of the command succeeds. In these rules, i, f, m and r range through *Id*, *F*, *M*, and *R*, respectively.

If it is not possible to derive a proof tree, then the entire Vitruv_I specification is invalid. We can distinguish three different cases here:

1. Access to not properly declared items, such as not defined values (7.14) or double declarations (e.g. in (7.20)).
2. Updates to fuzzy values (7.29) or constraints (7.28), which do not fit.
3. Updates to the *holds*-relation resulting in an empty set of interval relation (7.31).

In a proper translation process from Vitruv_L to Vitruv_I, the first error case should not appear, because it can be avoided by syntactical analysis. The other two cases are related to the semantics of the specification and occur, if the origin Vitruv_L specification is also invalid. Further invalidity checks are discussed in sec. 12.1 on page 255.

7.3.3.1. Non-modifying Semantics

Non-modifying semantics are used for literals and read access to values. The rules for literals (7.12, 7.13) are axioms in Vitruv_I, because they do not depend on the current state or environment. Reading access to values of identifiers (7.14) does not modify the current state, but, in contrast to literals, the values depend on the current state. In rule (7.15), the expression $f = m(f_1)$ means, that modifier m is applied to fuzzy set f_1 yielding fuzzy set f.

[fuzzy literal]

$$\frac{}{E, S \vdash f \models \langle E, S, f \rangle} \tag{7.12}$$

[modifier literal]

$$\frac{}{E, S \vdash m \models \langle E, S, m \rangle} \quad m \in \{\text{'not'}, \text{'norm'}, \text{'very'}, \text{'mol'}, \text{'somewhat'}, \text{'plus'}, \text{'extremely'}, \text{'intensify'}, \text{'slightly'}, \text{'above'}, \text{'below'}\} \tag{7.13}$$

[get value]

$$\frac{E(i) \in S.fuzzyvalue \quad E(i) \in Dom(S.value) \quad f = S.value(E(i))}{E, S \vdash \text{'value'}\ i \models \langle E, S, f \rangle} \tag{7.14}$$

[apply modifier]

$$\frac{E, S \vdash e \models \langle E, S, f_1 \rangle \quad E(i) \in Dom(S.modifier) \quad m = S.value(i) \quad f = m(f_1)}{E, S \vdash \text{'apply'}\ i\ e \models \langle E, S, f \rangle} \tag{7.15}$$

[binary expression]

$$\frac{E, S \vdash e_1 \models \langle E, S, f_1 \rangle \quad E, S \vdash e_2 \models \langle E, S, f_2 \rangle \quad f = \begin{cases} f_1 \cup f_2 & binop = \text{'or'} \\ f_1 \cap f_2 & binop = \text{'and'} \end{cases}}{E, S \vdash e_1 \; binop \; e_2 \models \langle E, S, f \rangle} \tag{7.16}$$

Relation literals are evaluated to a singleton set containing only the literal. This is needed to construct the set of relations used in the *holds*-relation. In rule (7.19) we define that the value of term *rel* is the same as of its constituting term *rs*.

[relation literal]

$$\frac{}{E, S \vdash r \models \langle E, S, \{r\} \rangle} \quad r \in \{\text{'='}, \text{'m'}, \text{'mi'}, \text{'f'}, \text{'fi'}, \text{'s'}, \text{'si'}, \text{'a'}, \text{'b'}, \text{'d'}, \text{'c'}, \text{'o'}, \text{'oi'}\} \tag{7.17}$$

[intermediate relation set]

$$\frac{E, S \vdash rs_1 \models \langle E, S, p_1 \rangle \quad E, S \vdash rs_2 \models \langle E, S, p_2 \rangle}{E, S \vdash rs_1 \text{ ','} \; rs_2 \models \langle E, S, p_1 \cup p_2 \rangle} \tag{7.18}$$

[relation set]

$$\frac{E, S \vdash rs \models \langle E, S, p \rangle}{E, S \vdash \text{'\{'} \; rs \; \text{'\}'} \models \langle E, S, p \rangle} \tag{7.19}$$

7.3.3.2. Declarations

Declarations share a common structure, requiring that identifier i is currently unbound. A new references for i is allocated with *new*, followed by an update of the appropriate components in S, either *interval*, *fuzzyvalue* or *block*.

[declare interval]

$$\frac{i \notin Dom(E) \quad (E', S'') = new(i, E, S) \quad S' = S'' \oplus (S''.interval \cup \{E'(i)\})\}}{E, S \vdash \text{'interval'} \; i \models \langle E', S', ok \rangle} \tag{7.20}$$

[declare fuzzy value]

$$\frac{i \notin Dom(E) \quad (E', S'') = new(i, E, S) \quad S' = S'' \oplus (S''.fuzzyvalue \cup \{E'(i)\})\}}{E, S \vdash \text{'fuzzyvalue'} \; i \models \langle E', S', ok \rangle} \tag{7.21}$$

[declare modifier]

$$\frac{i \notin Dom(E) \quad (E', S'') = new(i, E, S) \quad S' = S'' \oplus (S''.modifier \cup \{E'(i)\})\}}{E, S \vdash \text{'modifier'} \; i \models \langle E', S', ok \rangle} \tag{7.22}$$

[declare block]

$$\frac{i \notin Dom(E) \quad (E', S'') = new(i, E, S) \quad S' = S'' \oplus (S''.block \cup \{E'(i)\})\}}{E, S \vdash \text{'block'} \; i \models \langle E', S', ok \rangle} \tag{7.23}$$

7.3.3.3. Extended Declarations

Extended declarations define functions between intervals, elements, blocks, etc. The rules are similar, as an example we discuss rule (7.26). A declaration inBlock i_1, i_2 states that interval i_1 is in block i_2 (cf. tables 7.1 on page 132 and 7.2 on page 136). We need a proper environment E and state S, where identifiers i_1 and i_2 are declared in E and their respective references $E(i_1)$ and $E(i_2)$ belong to their proper sets in S, i.e. i_1 is an interval and i_2 is a block. To prevent double declarations we demand that each interval is only mapped once to a block, hence $E(i_1)$ is not allowed in the domain of *S.inBlock*. If these precondition are satisfied, we update the *inBlock* component of S by adding the pair $E(i_1) \mapsto E(i_2)$ resulting in a new state S'. Environment E is not modified, hence we have $\langle E, S', ok\rangle$ is the result of declaration inBlock i_1, i_2. Analogously, in rule (7.24) we check that i_1 is already declared as fuzzy value and that i_1 is not yet declared as an element, before updating *S.element*. In rule (7.25), we require that i_1 is declared as an element of i_2, before updating *S.length*.

$$\text{[element]}\quad \frac{\begin{array}{ll} E(i_1) \in S.fuzzyvalue & E(i_2) \in S.interval \\ E(i_1) \notin Dom(S.element) & S' = S \oplus (S.element \oplus \{E(i_1) \mapsto E(i_2)\}) \end{array}}{E, S \vdash \text{`element'}\ i_1\text{`,'}\ i_2 \models \langle E, S', ok\rangle} \tag{7.24}$$

$$\text{[length]}\quad \frac{\begin{array}{c} S.element(E(i_1) = E(i_2) \quad E(i_2) \notin Dom(S.length) \\ S' = S \oplus (S.length \oplus \{E(i_2) \mapsto E(i_1)\}) \end{array}}{E, S \vdash \text{`length'}\ i_1\text{`,'}\ i_2 \models \langle E, S', ok\rangle} \tag{7.25}$$

$$\text{[in block]}\quad \frac{\begin{array}{ll} E(i_1) \in S.interval & E(i_2) \in S.block \\ E(i_1) \notin Dom(S.inBlock) & S' = S \oplus (S.inBlock \oplus \{E(i_1) \mapsto E(i_2)\}) \end{array}}{E, S \vdash \text{`inBlock'}\ i_1\text{`,'}\ i_2 \models \langle E, S', ok\rangle} \tag{7.26}$$

7.3.3.4. Value Updates and Constraint Definitions

Value updates and constraint definitions apply to fuzzy values and modifiers. We need to check that constraints and current assignments fit. This check is performed by predicate *checkConstraint*, requiring that value and constraint fit if both are defined:

$$checkConstraint(E, S, i) = \begin{cases} false & \text{if } i \notin Dom(E) \\ true & \text{if } i \in Dom(E) \wedge (E(i) \notin Dom(S.constraint)) \\ true & \text{if } i \in Dom(E) \wedge (E(i) \notin Dom(S.value)) \\ f_1 \subseteq f_2 & \text{otherwise} \end{cases} \tag{7.27}$$

where

$$f_1 = S.value(E(i))$$
$$f_2 = S.constraint(E(i))$$

With predicate *checkConstraint* we can define the rules for constraint and value updates of fuzzy values. For modifiers we do not need any additional predicates.

$$\frac{\begin{array}{cc} E, S \vdash e \models \langle E, S, f \rangle & E(i) \in Dom(S.fuzzyvalue) \\ S' = S \oplus (S.constraint \oplus \{E(i) \mapsto f\}) & checkConstraint(E, S', i) \end{array}}{E, S \vdash \text{'constraint'}\ i\ \text{','}\ e \models \langle E, S', ok \rangle}\ [\text{constraint}] \tag{7.28}$$

$$\frac{\begin{array}{cc} E, S \vdash e \models \langle E, S, f \rangle & E(i) \in Dom(S.fuzzyvalue) \\ S' = S \oplus (S.value \oplus \{E(i) \mapsto f\}) & checkConstraint(E, S', i) \end{array}}{E, S \vdash \text{'update'}\ i\ \text{','}\ e \models \langle E, S', ok \rangle}\ [\text{update fuzzy value}] \tag{7.29}$$

$$\frac{\begin{array}{cc} E, S \vdash e \models \langle E, S, m \rangle & E(i) \in Dom(S.modifier) \\ S' = S \oplus (S.value \oplus \{E(i) \mapsto m\}) & \end{array}}{E, S \vdash \text{'update'}\ i\ \text{','}\ e \models \langle E, S', ok \rangle}\ [\text{update modifier}] \tag{7.30}$$

7.3.3.5. The Holds Relation

The *holds* relation is the centerpiece of Vitruv_I. Each holds-statement defines a set, p_1, of relations which shall hold between two intervals (see rule (7.19)). This set of relations is regarded as a set of mutually exclusive relations, i.e. one of these relations shall hold, but we do not know which one. If we have already defined another set, p_2, of relations holding between these two intervals, then we have to calculate the intersection p_3 of p_1 and p_2, because the combination of independent hold-statements is always a conjunction of the relations. The resulting p_3 is not allowed to be empty, because this would be an inconsistent specification. In this way, the holds-statement is not a simple assignment overriding previous values, but depends on values previously assigned.

$$\frac{\begin{array}{ccc} E(i_1) \in Dom(S.interval) & E(i_2) \in Dom(S.interval) & E, S \vdash rel \models \langle E, S, p_1 \rangle \\ p_3 \neq \emptyset & \multicolumn{2}{c}{S' = S \oplus (S.holds \oplus \{(E(i_1), E(i_2)) \mapsto p_3\})} \end{array}}{E, S \vdash \text{'holds'}\ i_1\ \text{','}\ i_2\ \text{','}\ rel \models \langle E, S', ok \rangle}\ [\text{holds}] \tag{7.31}$$

with

$$p_3 = \begin{cases} p_1 & \text{if } (E(i_1), E(i_2)) \notin \mathit{Dom}(S.\mathit{holds}) \\ p_1 \cap p_2 & \text{otherwise and additionally } p_2 = S.\mathit{holds}(E(i_1), E(i_2)) \end{cases}$$

7.3.3.6. Statements

Statements are combined in the usual way. The abstract syntax is ambiguous, because a sequence of statements can have different syntax trees. But this ambiguity is resolved in the semantics, because we enforce here a strict left-to-right evaluation.

$$[\text{statement}] \quad \frac{E, S \vdash \mathit{term} \models \langle E', S', v \rangle}{E, S \vdash \mathit{term}\,\text{`;'} \models \langle E', S', v \rangle} \tag{7.32}$$

$$[\text{sequence}] \quad \frac{E, S \vdash \mathit{statement}_1 \models \langle E', S', v' \rangle \quad E', S' \vdash \mathit{statement}_2 \models \langle E'', S'', v'' \rangle}{E, S \vdash \mathit{statement}_1\ \mathit{statement}_2 \models \langle E'', S'', v'' \rangle} \tag{7.33}$$

7.4. Linearizing Vitruv_L

The linearizing of a Vitruv_L specification means translation of Vitruv_L to Vitruv_I. We can separate the translation process in an allocation and an application phase, occurring intertwined but clearly distinguishable. The translation process is a syntax-driven translation (Aho et al., 1987, chap. 5), based on attributed syntax trees as derived by the static semantics (cf. sec. 6 starting on p. 95).

The core mechanism of the linearization is as follows. Each first entity occurrence in the binding results in an allocation. After an entity is allocated, an application of rules, constraints and value updates using the entity may follow. Allocation means introducing new identifiers for entities resolving the nested name-space into a flat one, which is similar to the allocation of storage cells when explaining the semantics of programming languages. For objects consisting of other objects this process is applied recursively, resulting in a pre- and post-order traversal through the nesting hierarchy. Similar to the static semantics of Vitruv_L and the operational semantics of Vitruv_I we need a dynamic environment mapping Vitruv_L identifiers to their Vitruv_I counterparts. In particular, this is important for resolving nested scopes and the rebinding of values in a context.

We will discuss now details of the linearization, starting with fuzzy types, followed by compound relations and finishing with classes.

7.4.1. Fuzzy Types

Fuzzy types define modifiers and named fuzzy values. The binding of a fuzzy type requires allocation of new fuzzy values and modifiers, followed by value updates.

In spec. 7.2 we show the translation of the binding of fuzzy type Brightness (see spec. 5.12 on page 84). At first we translate the type declaration in a series of declaration statements together with the initial values of the modifiers. It is followed by the assignments of terms and modifiers made in the binding. We prefix the identifiers of terms and modifiers here with br_ for the sake of easier identification. In an automatic translation process we would simply use numbered identifiers to ensure that no naming conflicts can arise.

Specification 7.2 Fuzzy Type Brightness in $Vitruv_l$

```
fuzzyvalue (br_black).
fuzzyvalue (br_dark).
fuzzyvalue (br_muddy).
fuzzyvalue (br_shining).
fuzzyvalue (br_bright).
fuzzyvalue (br_white).
modifier (br_very).
modifier (br_not).
modifier (br_more_or_less).
update (br_very,          very).
update (br_not,           not).
update (br_more_or_less,  mol).
update (br_black,         trapezoid (0, 0, 10, 20)).
update (br_dark,          triangle (10, 25, 40)).
update (br_muddy,         triangle (25, 40, 120)).
update (br_shining,       triangle (100, 170, 190)).
update (br_bright,        triangle (170, 190, 240)).
update (br_white,         trapezoid (230, 240, 255, 255)).
update (br_very,          extremely).
```

Re-binding of Brightness's terms and modifiers in some context requires fresh values without destroying the old values. Hence, we introduce new declarations and value updates of for re-bound fuzzy values and modifiers. Additionally we have to modify the dynamic environment mapping the re-bound identifiers in $Vitruv_L$ to the new allocated identifiers in $Vitruv_l$. As an example consider a re-binding of term black to value zfunction (0, 20). In spec. 7.3 on the next page we use identifier br_black1 to denote the re-bound term black.

Specification 7.3 Re-binding term black in $Vitruv_I$

```
fuzzyvalue (br_black1).
update (br_black1, zfunction (0, 20)).
```

7.4.2. Compound Relations

Compound Relations are not mentioned in the binding, however, they are used in the body of classes. Their translation is a classical macro expansion mechanism, relying on the current binding of fuzzy type DURATION, as discussed in sec. 5.8.5 on page 84. We will discus the translation of the body together with the body of classes, because all rules within a compound relation are also allowed in class bodies.

Locally declared elements in compound relations are instances of class Interval. Hence their translation into $Vitruv_I$ follows exactly those of elements declared within a class. We have only to be careful that for each application of a compound relation all of its local elements are again declared as fresh class instances in $Vitruv_I$.

7.4.3. Classes

A class in $Vitruv_L$ consists of two parts: the class definition, defining structure and behavior, and the binding, giving each instance its individual values. Consequently, we have to consider both parts in the translation process: each class instance is a set of intervals and a set of fuzzy values, all of them following the structure of the class definition. We do not consider events as a special kind of elements here, but use only the event's enabling interval, because in $Vitruv_I$ we consider only the static structure. Dynamic aspects introduced by events are handled in sec. 8 on page 149. Similarly, loops and selectors are only considered as unstructured intervals.

We show as examples how to translate some of the specifications presented in sec. 5.8.7, starting at page 86. The classes used are specified in spec. 5.13 on page 87. For each example translation the respective binding specifications already presented in sec. 5.8.7 are repeated here, for the reader's convenience.

The linearization process for classes is driven by the binding. For each object x, instantiated by object x : X ... end we apply an allocation phase for x and all its fuzzy values, followed by the recursive linearization of the elements of x. After the allocation phase the application phase follows, handling the body of x, i.e. defining the constraints and relations of x and its elements.

The allocation phase of object x results in $Vitruv_I$ statements identifying x as an interval (interval(x)). Each fuzzy value y of x is identified as fuzzy value by fuzzyvalue (y) and as an element of x by element (y, x). The length element L is additionally identified by length (x, L). If and only if the attributes of x occur in the binding, their linearization will be triggered. For each attribute a of x, with a occurring in the binding, we

have the relation that x contains, or is started or is finished by a, denoted by holds (x, a, { c, si, fi }). For private and inherited elements we apply the same scheme. The special attributes alpha and omega never occurring in the binding are handled in the application phase.

In spec. 7.4 we show the allocation phase for the binding of a1 and a, respectively. For easier identification we prefix the elements of a1 and a with a1_ and a_, respectively. Their class A consists only of an interval a, used in the binding of a1. The allocation of a1 states interval a1 and the inherited elements a1_length, a1_alpha and a1_omega from class Interval. In a1 we bind also object a, allocating again the same structure as for a1. The binding recurses not into object a, therefore we do not enter element a_a and do not allocate further elements.

Specification 7.4 Allocation phase for assigning large to the length of a.

```
object a1 : A
  object a : A
    let length := DURATION.large;
  end;
end;
```

```
interval    (a1).          // now entering a1, declaring all elements
fuzzyvalue  (a1_length).
element     (a1_length, a1).
length      (a1, a1_length).
interval    (a1_alpha).
interval    (a1_omega).

interval    (a).           // now entering a, declaring all elements
fuzzyvalue  (a_length).
element     (a_length, a).
length      (a, a_length).
interval    (a_alpha).
interval    (a_omega).
holds       (a1, a, { c, si, fi }). // a is an attribute of a1

                           // ... but we do not enter a_a!
```

The application phase considers the body of classes and assignments in the binding. Each constraint a is c and each interval relationship a r b in the body results in the corresponding Vitruv$_l$ statements constraint (a, c) and holds (a, b, { r }). For compound relations we additionally have to introduce for each local element e a new

fresh interval by interval (e) each time the relation is applied.

While these translations are straightforward, we have also to deal with some peculiarities: nil objects and class Interval. For nil objects we apply the rules given on p. 64:

1. we consider only relations where both parameters are not nil,
2. constraints with a not existing l-value are ignored,
3. in constraint expressions, not existing values are set to function with constant value 1, i.e. their value is linear((-inf, 1.0), (inf, 1.0)).

Class Interval consists of atomic intervals alpha and omega, they are available in each class. Because they are allowed to appear in the binding, we do include them explicitly as intervals in the linearization process, as already seen in spec. 7.4 on the preceding page. Their specific rules, in particular that they are not activated during the current interval but rather start and finish their embedding interval, are stated in the application phase of the class linearization.

In spec. 7.5 on the next page, we show the application phase, the second half of the linearization, for our example started in spec. 7.4 on the preceding page. We have three different parts there: the rules and constraints for alpha and omega of objects a and a1, resp.; the constraint for a; and finally the binding of a updating the value of fuzzy value a.length. The constraint declared in class A is ignored for object a, because the constraint refers its attribute a, which is not instantiated in the binding.

7.4.4. The Prelude

The prelude and its binding provide a standard set of definitions, which can be used in Vitruv_L specifications without the need to mention them explicitly. The binding of the prelude provides default values for the elements defined in the prelude.

Concerning the linearization of Vitruv_L the prelude and its binding poses no difficulties, we have only to use the definitions from the prelude including its binding, if in the Vitruv_L specification references are made to the prelude definitions. Typical applications of the prelude are (implicit) references to class Interval or to type DURATION. In the examples in this sections these references are already resolved , e.g. in spec. 7.5 on the next page, where among other instance a_alpha of class Interval is linearized.

7.4.5. Blocks: Dealing with Loops, Selectors and Scenes

The examples presented so far are rather simple, neither scenes nor selectors and loops occurred. We now deal with them.

Specification 7.5 Application phase for assigning large to the length of a.

```
object a1 : A
  object a : A
    let length := DURATION.large;
  end;
end;
```

```
holds (a, a_alpha, { si }).       // rules declared for a, which are
holds (a, a_omega, { fi }).       // inherited from Interval
constraint (a_alpha, value(DU_zero)).
constraint (a_omega, value(DU_zero)).
constraint (a_length, rectangle (0.0, inf) .
// no constraint for a_a, because it does not exist

update (a_length, value (DU_large)).  // binding of a

holds (a1, a1_alpha, { si }).      // rules declared in a1, which are
holds (a1, a1_omega, { fi }).      // inherited from Interval
constraint (a1_alpha, value(DU_zero)).
constraint (a1_omega, value(DU_zero)).
constraint (a1_length, rectangle (0.0, inf) .

constraint (a_length, value (DU_large)).  // rule in a1 concerning a
                                   // no further binding of a1
```

Scenes, selectors and loops differ from other intervals, because they introduce a block, i.e. a set of intervals which has to be consistent, but with no relations to intervals outside the block. It is sufficient to declare only the root interval r of the block b by inBlock (r, b), because all other intervals inside block b can be deduced by a reachability analysis following the declared interval relations.

We have two patterns for declaring blocks in Vitruv$_I$ if we translate an Vitruv$_L$ specification, the first for scenes and the second for loops and selectors. For scenes, we have already an allocation because scenes are classes resulting in an interval for the scene. This interval is the root of the new block, therefore we have only to declare a new block and state that the interval for the scene is a member of the block.

The pattern for loops and selectors is slightly different. They result in an interval i, which is used in the declaring block and thus is not the root of the block introduced by loops and selectors. The interval relations between *interval* i and the respective event are only implicitly declared in Vitruv$_L$ and are now made explicit, i.e. we declare the relations s or si or = hold between the enabling interval of the event and interval i for each loop or selector. For each loop body and for each selector path, we have to allocate a new interval, which becomes the root of the new block. After that, we follow the pattern for scenes, declaring a new block and stating that the root of the block is a member of the block. Since occurrence intervals of events are not used in Vitruv$_I$, we have no relations between the event and the intervals for loop bodies and selector paths.

In spec. 7.6 we show an example for both patterns, first scene the_scene, then loop the_loop, which depends on event the_event. Both patterns start with the allocation of the interval for the declared element in Vitruv$_L$, i.e. interval (the_scene) and interval (the_loop). For the loop, we declare the relations between the event and interval the_loop, after that we allocate an interval for the loop body (interval (the_loop_body)), which becomes the root of the new block for the loop. It is followed by an allocation of new blocks (the_scene_block and the_loop_body_block). Blocks and their root intervals are then connected by statements inBlock (the_scene, the_scene_block) and inBlock (the_loop_body, the_loop_body_block).

Specification 7.6 Declaring blocks in Vitruv$_I$

```
interval (the_scene).
block (the_scene_block).
inBlock (the_scene, the_scene_block).

interval (the_loop).
holds (the_event, the_loop, { s, si, = }).
interval (the_loop_body).
block (the_loop_body_block).
inBlock (the_loop_body, the_loop_body_block).
```

8. Vitruvian Nets

After defining the static semantics and the semantics for the event-free behavior of Vitruv_L in the preceding two chapters, we now move to the full dynamic semantics of Vitruv_L including in particular the event-based behavior. We do this with a Petri net variant we call *Vitruvian Nets*, abbreviated by VN. The dynamic semantics of Vitruv_L has to address different aspects:

Vitruvian Nets

- the static structure of temporal relationships between intervals,
- the dynamic flow of control originating by events applied in loops and selectors, and finally
- the combination of static structure and dynamic behavior.

Realizing a divide-and-conquer approach, we model these three aspects independently (as far as possible) as different Petri net variants, thereby reducing the complexity of each variant. We start with a common base called abstract Vitruvian Nets, defining the common structure and behavior of the forthcoming Petri net variants (sec. 8.1). It is followed by a discussion of temporal relationships in sec. 8.2 on page 153, introducing Petri nets with fuzzy timing, and the control flow definition in sec. 8.3 on page 163, applying Petri nets with fuzzy markings. In sec. 8.4 on page 168 we combine this net type to Basic Vitruvian Nets dealing with events, loops, and selectors. For scenes we extend Basic Vitruvian Nets to general Vitruvian Nets (VN), defined in sec. 8.5 on page 178. The hierarchy of these different kinds of Vitruvian Nets is shown in fig. 8.1 on the following page. Finally, in sec. 8.6 on page 183, we discuss how to compose a various scenes together resulting in a translation process for Vitruv_L specification into Vitruvian Nets.

General definitions and notations for Petri nets and multi sets can be found in sec. B.2 on page 288.

8.1. Prelimaries: Abstract Vitruvian Nets

Abstract Vitruvian Nets are the basis definition of Vitruvian Nets (VN). They define the common structure and behavior, however, most of the definitions need to be refined in derived nets. Vitruvian Nets are colored high-level Petri nets (Jensen, 1997; Smith, 1998), accordingly each place has a color set or type, and tokens are individuals.

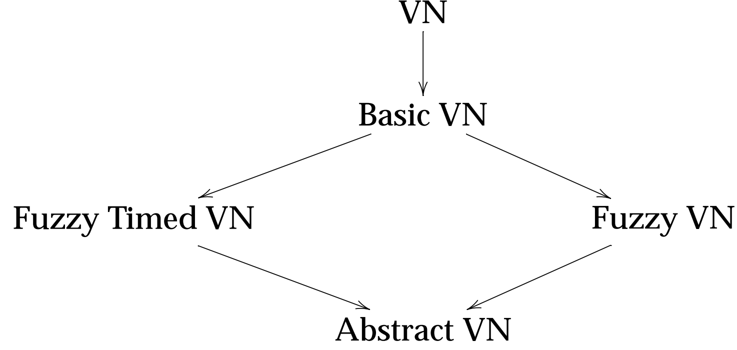

Figure 8.1.: The Hierarchy of Vitruvian Nets

abstract Vitruvian Net

Definition 8.1 (Abstract VN) *An* abstract Vitruvian Net *is a triple* (N, Σ, C) *defined as:*

1. $N = (T, P, A)$ *is a Petri net.*
2. Σ *is a finite set of types or, equivalently, a finite set of color sets.*
3. $C : P \to \Sigma$ *is a color function, assigning to each place a type.*

$\mathcal{A}$

The set of all abstract Vitruvian Nets is denoted with $\mathcal{A}$*.*

Remarks:

Types (or color sets) here may represent types we also have in Vitruv_L, thus we realize each type as a set of values. Hence Σ is a set of sets and color function C maps each place $p \in P$ to an element of Σ, i.e. a set of values.

The marking of an abstract VN is a multi-set of places and tokens, thereby guaranteeing that multiple tokens with the same value can exist at a place. In the literature (e.g. Baumgarten (1996)) such tokens are called individual tokens. We need this property to model places holding all event values of an enabled event. Certainly, these event values might be equal and we want to differentiate between them. Hence, a simple set of token values is not appropriate, and thus we use a multi-set as also found in the literature (Jensen, 1997; Smith, 1998). Each token of place p is an element of the type $C(p) \in \Sigma$.

Definition 8.2 (Marking) *The set of all possible tokens, W, of an abstract VN* (N, Σ, C) *is defined as*

$$W = \{(p, v) \mid p \in P, v \in C(p)\}, \tag{8.1}$$

marking

the marking*, M, is a multi-set over W:*

$$M \in \mathcal{M}(W). \tag{8.2}$$

The marking of a specific place, p, is notated as map evaluation, $M(p)$, yielding a multi-set of token values. This construction is similar to relation images, but here the result is a multi-set and not a regular set.

Definition 8.3 (Marking of a place) *Let M be a marking and p be a place of an abstract VN. The marking of place p is defined as the multi-set*

$$M(p) = \{(v, n) \mid ((p, v), n) \in M\}. \tag{8.3}$$

The firing behavior defines when and how tokens are removed from and added to places. In contrast to the approach of Jensen (1997), we do not consider arc inscriptions explicitly, but use a rather abstract definition. The firing behavior of each transition, t, can be seen as a function mapping possible markings in the pre-set, $\bullet t$, of t to their corresponding markings in the post-set, $t\bullet$, of t. The firing behavior of the entire net is then the union of the firing behavior of each transition, defined as a map from transitions to functions between markings.

Definition 8.4 (Firing Function) *If transition $t \in T$ in a VN fires, the token movement $F(t)$ for t is a partial function, where the* firing function *F is a second order function:*

firing function

$$F : T \to (\mathcal{M}(W) \nrightarrow \mathcal{M}(W)), \tag{8.4}$$

where $A \nrightarrow B$ denotes the partial functions from A to B.

In the following, we present some general properties of F. However, for abstract VN the definition of F remains abstract. The derived net variants in the following sections define specific instances of F.

The usual firing behavior in Petri nets depend on pre- and post-sets of transitions: if transition t fires, only tokens from $\bullet t$ are removed and added to $t\bullet$. Situations are triplets of transition t and markings w_i and w_o, describing possible firings. Thus, a firing function F collects of all possible situations.

Definition 8.5 (Situation) *Let (T, P, A) be an abstract VN, and $w_i, w_o \in \mathcal{M}(W)$. A* situation *is a triple (t, w_i, w_o) representing a possible firing of transition t relative to a firing function F with $F(t)(w_i) = w_o$. For each situation $s = (t, w_i, w_o)$ markings w_i and w_o can only denote places and tokens from the pre- and post-set of t, resp., i.e. we require that*

situation

$$w_i \subseteq_{ms} \{(p, v) \mid (p, v) \in W \wedge p \in \bullet t\}_{ms} \tag{8.5}$$

$$w_o \subseteq_{ms} \{(p, v) \mid (p, v) \in W \wedge p \in t\bullet\}_{ms} \tag{8.6}$$

The set of all situations $\mathcal{S}$ is defined as

$\mathcal{S}$

$$\mathcal{S} = \{(t, w_i, w_o) \mid t \in \text{dom}(F), w_i \in \text{dom}(F(t)), w_o \in \text{ran}(F(t)), w_o = F(t)(w_i)\}. \tag{8.7}$$

Definition 8.6 (Abstract Firing Behavior) *Let $M_1, M_2, w_i, w_o \in \mathcal{M}(W)$ be markings, $t \in T$, and $F(t)(w_i) = w_o$, i.e. (t, w_i, w_o) is a situation. If transition t fires in situation (t, w_i, w_o) and marking M_1 with $w_i \subseteq_{ms} M_1$, the new marking M_2 is defined as the formal sum*

$$M_2 = (M_1 - w_i) + w_o. \tag{8.8}$$

Remarks:

1. Multi-sets w_i and w_o are essentially the arc expressions for the pre- and post-set of transition t, resp., because w_i and w_o denote the multi-sets of tokens removed from the pre-set and added to the post-set.
2. Often $F(t)$ maps only one value w_i to one value w_o, thereby yielding a unique behavior of t. In particular, this is the case in nets with anonymous tokens, as shown in the next example.

Example 8.1 Consider the abstract VN in fig. 8.2 on the next page. It models as P/T-net with arc weights 2 for arc (p_1, t_1), 3 for arc (p_2, t_1), and 1 for arc (t_1, p_3). As a P/T-net we have only one color set with exactly one value: $\star$. Thus, the formal definition of this net is

1. $T = \{t_1\}$,
2. $P = \{p_1, p_2, p_3\}$,
3. $A = \{(p_1, t_1), (p_2, t_1), (t_1, p_3)\}$,
4. $\Sigma = \{\{\star\}\}$,
5. $C = \{(p_1, \{\star\}), (p_2, \{\star\}), (p_3, \{\star\})\}$.

Firing function F is defined according to the arc weights as

$$F = \{(t, \{\{((p_1, \star), 2), ((p_2, \star), 3)\} \mapsto \{((p_3, \star), 1)\}\})\}.$$

□

For the sake of completeness, we can collect an abstract VN together with its firing function and initial marking to define a an abstract Vitruvian system. However, such systems are not intended to be used directly, but serve as the basis for derived systems defined in the following sections.

abstract Vitruvian Net system

Definition 8.7 (Abstract VN System) *An* abstract Vitruvian Net system *is a five-tuple (N, Σ, C, F, M_0) consisting of an abstract VN (N, Σ, C), firing function F and an initial marking M_0.*

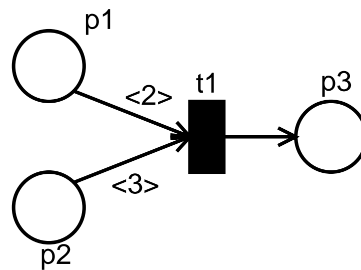

Figure 8.2.: Simple P/T-net with arc-weights as abstract VN.

The definitions of abstract VN and abstract VN systems do not deal with all elements we usually find in (high-level) Petri nets. We list the differences here and take care of many of them in the next sections defining derived Vitruvian Nets.

- We do not define the enabling conditions of transitions preventing the direct use of abstract VN systems, because we define only what happens when a transition occurs, but do not define which conditions have to be satisfied to enable the transition for firing. Such enabling conditions are left to derived VN. Also guard conditions, as used in other high-level nets such as Colored Petri Nets (Jensen, 1997) or Pr/T-nets (Smith, 1998), are not mentioned. However, generally, they are part of the enabling conditions.
- Capacity restrictions, i.e. the number of tokens that are allowed in a place, are not enforced. They might appear in derived nets, otherwise the capacity is unlimited.
- There are no net structure constraints, such as acyclic nets. Such constraints can be imposed in derived nets.

8.2. Vitruvian Nets with Fuzzy Timing

Petri nets with fuzzy timings are suitable to model the intervals and their relations we found in Vitruv_L. Fuzzy timing Petri nets were presented first by Murata (1996), the definition matured in publications by Murata et al. (1999) and by Zhou and Murata (1999). We adopt the latter model for the use within Vitruvian Nets.

8.2.1. Formal Definition

Fuzzy Timing Vitruvian Nets (FTVN) introduce a timed behavior based on possibility distributions of time-points. These possibility distributions are disjunctive (Dubois and Prade, 1989), i.e. only one time-point of their support will actually happen, but

which one is not known. Dubois and Prade call this time-pont to be *ill-known*. The possibility distribution denotes the possibility of this ill-known time-point.

$\mathcal{T}$
$\mathcal{D}$

Definition 8.8 *Let $\mathcal{T}$ be our time scale, i.e. $\mathcal{T} = \mathbb{R}_0^+$, and $\tau \in \mathcal{T}$ be an element of the time scale. The set of all fuzzy sets on the time scale is denoted with $\mathcal{D}$, $\mathcal{D} = \mathcal{F}(\mathcal{T})$. Each $d \in \mathcal{D}$ is characterized by its membership function $\mu_d(\tau)$.*

We often need two specific durations, one denoting duration 0, the other denoting that no duration exists at all.

$d_\perp$
d_0

Definition 8.9 *The empty duration $d_\perp \in \mathcal{D}$ denotes that no duration is possible, the zero duration $d_0 \in \mathcal{D}$ denotes the exact length of zero. Both are defined by their membership functions $\mu_{d_\perp}$ and μ_{d_0}:*

$$d_\perp : \mu_{d_\perp}(\tau) = 0 \quad \text{for all } t \in \mathcal{T} \tag{8.9}$$

$$d_0 : \mu_{d_0}(\tau) = \begin{cases} 1 & \text{if } \tau = 0 \\ 0 & \text{otherwise} \end{cases} \tag{8.10}$$

Structurally, FTVN extend abstract VN by assigning fuzzy delays to arcs from transitions to places. Each token carries a timestamp, which is updated with each firing by the respective delays. We do not need any data except the timestamps as tokens, because we only want to model the activity of an interval (details in sec. 8.2.2 on page 159). Therefore, it is sufficient to restrict the set of types to the Cartesian product of the set with the only value $\star$ and the set of durations, $\mathcal{D}$.

FTVN

Definition 8.10 *A* FTVN *is a four-tuple $(N, \Sigma, C, delay)$ where (N, Σ, C) is an abstract VN. Additionally we have*

1. *$delay : (T \times P) \nrightarrow \mathcal{D}$ is the delay for each outgoing arc of $t \in T$, i.e.* $\mathrm{dom}(delay) = \{(t, p) \mid t \in T, p \in t\bullet\}$.
2. *$\Sigma = \{\{\star\} \times \mathcal{D}\}$, i.e. all tokens are pairs of the anonymous value, $\star$, and a fuzzy time value, d.*

$\mathcal{T}$

The set of all FTVN is denoted with $\mathcal{T}$.

For the semantics of a FTVN $(N, \Sigma, C, delay)$ we need the usual markings, $M \in \mathcal{M}(W)$. For defining the enabling conditions of a transition we need an untimed marking, which is the projection of M to the token value to without last component, the fuzzy timestamp d. We have to preserve the multiplicity, hence we have to sum up the number of all tokens in each place to calculate the new multiplicity of the untimed marking.

M^u

Definition 8.11 (Untimed Marking) *Let $M \in \mathcal{M}(W)$ be a timed marking. The untimed marking M^u of M is defined as the formal sum of all untimed elements of M:*

$$M^u = \sum_{m \in M} \{\mathrm{pr}_{FTVN}(m)\} \tag{8.11}$$

where pr_{FTVN} *is the projection of a timed element of M to an untimed element*

pr_{FTVN}

$$\mathrm{pr}_{FTVN} : ((P \times (\{\star\} \times \mathcal{D})) \times \mathbb{N}) \to ((P \times \{\star\}) \times \mathbb{N}) \tag{8.12}$$

$$((p, (\star, d), n) \mapsto ((p, \star), n) \quad \textit{for some } p \in P, d \in D, n \in \mathbb{N} \tag{8.13}$$

Remarks:

1. We need the formal sum to get the number of all tokens in each place p, independent of the fuzzy timestamps of the tokens.
2. Let $m = ((p, \star, d), n) \in M$, then the expression $\{\mathrm{pr}_{FTVN}(m)\}$ is a multi-set with element $(p, \star)$ occurring n times.

We give now details of the functions defined by F. Thus, we will define when a transition is enabled and occurs, and also define how timestamps, the tokens' value, are calculated. This is worked out in the next definitions.

Transition t or situation s is quasi-enabled, if there are enough tokens available in $\bullet t$, independent of their timestamps.

Definition 8.12 (quasi-enabled) *A situation* $s = (t, w_i, w_o) \in \mathcal{S}$ *is* quasi-enabled *at a particular marking M if*

quasi-enabled

$$w_i^u \subseteq_{ms} M^u. \tag{8.14}$$

In FTVN, we establish a "first come, *possibly* first serve" fire policy, such that earlier enabled situations are preferred. The enabling time depends on the latest arrival of the token that quasi-enables the situation: this is the earliest time, at which the situation is enabled. For the occurrence time, we determine the earliest enabling time of all quasi-enabling situations of a transition. The intersection of the earliest enabling time of all quasi-enabled situations and the enabling time of a particular situation s becomes then the occurrence time of s.

Firstly, we define operator *latest* determines latest-arrival/lowest-possibility distribution, similar to extended maximum (B.39) of intervals, but the height is restricted to the minimal height of all fuzzy sets.

Definition 8.13 (latest) *Let* $d_1, d_2 \in \mathcal{D}$ *be fuzzy durations, h the minimum of the height of* d_1 *and* d_2*, i.e.* $h = \min(\mathrm{height}(d_1), \mathrm{height}(d_2))$*. Let m be the extended maximum of* d_1 *and* d_2 *(see eq.* (B.39)*) with membership function* $\mu_m(\tau)$*. Operator latest applied to* d_1 *and* d_2 *is defined by its membership function* $\mu_{latest}(\tau)$ *as:*

latest

$$\mu_{latest}(\tau) = \begin{cases} h & \mu_m(\tau) > h \\ \mu_m(\tau) & \textit{otherwise} \end{cases} \tag{8.15}$$

For more than two arguments, latest can applied recursively:

$$\textit{latest}(d_1, d_2, \ldots, d_n) = \textit{latest}(d_1, \textit{latest}(d_2, \ldots \textit{latest}(d_{n-1}, d_n))) \tag{8.16}$$

Remarks:

Murata and Zhou do not present a formal definition of *latest*. An algorithm for *latest* is presented by Zhou and Murata (1999), however, the algorithm is not usable for general fuzzy sets, but requiring all involved fuzzy sets having a trapezoidal shape only.

With the definition of operator *latest* we can now define how to calculate the fuzzy enabling time of a situation.

fuzzy enabling time

Definition 8.14 (Fuzzy Enabling Time) *The* fuzzy enabling time $e_s(\tau)$ *of a situation,* $s = (t, w_i, w_o) \in \mathcal{S}$*, is the possibility distribution of the latest arrival time among the arrival time of all tokens in* w_i*. If* w_i *has n tokens with timestamps* $d_i(\tau)$, $i \in \{1, \ldots, n\}$*, we have*

$$e_s = latest(d_1, \ldots, d_n) \tag{8.17}$$

Remarks:

The enabling time is defined for each quasi-enabled situation *s*. If there are several quasi-enabled situations for the same transition *t*, these situations have in general different enabling times.

The firing of a transition selects one enabled situation *s* and calculates its occurrence time. If we have several enabled situations, we prefer the earliest arrival time with highest possibility. Therefore, we determine the intersection between the enabling time of *s* and the earliest enabling time of all enabled situations including *s*.

fuzzy occurrence time

Definition 8.15 (Fuzzy Occurrence Time) *The* fuzzy occurrence time $o_s(\tau)$ *of a situation,* $s = (t, w_i, w_o) \in \mathcal{S}$*, is the possibility distribution of the time at which the situation s occurs (i.e. fires). Let there be m quasi-enabled situations of t (including s) with fuzzy enabling times* $e_k(\tau)$, $k \in \{1, \ldots, m\}$*, let* k_s *(*$1 \leq k_s \leq m$*) be the index of s in* $\{1, \ldots, m\}$ *. Then the fuzzy occurrence time of situation* $s = (t, w_i, w_o)$ *with enabling time* $e_s(\tau)$ *is defined by*

$$o_s(\tau) = \min\{e_s(\tau), earliest_{k \in \{1,\ldots,m\}}\{e_k(\tau)\}\} \tag{8.18}$$

Remarks:

1. Operator *earliest* determines the earliest-arrival/highest-possibility distribution and is the extended minimum (B.38) on intervals. Murata seems not to be sure about that, in his earlier publications (Murata, 1996; Murata et al., 1999) he defined *earliest* as the extended minimum, in a later publication (Zhou and Murata, 1999) *earliest* is described to be only similar to the extended minimum but without giving reasons for not being the extended minimum. As for operator *latest*, no formal definition is presented for *earliest*, except for an algorithm in the last publication. However, this algorithm is not usable for general fuzzy sets, but requiring all involved fuzzy sets having a trapezoidal shape.

2. The intersection of e_s and the earliest enabling time results in applying the min-operator.
3. If the earliest enabling time and the enabling time of situation s have a disjoint support, then s cannot occur, because its occurrence time is $o_s = d_\perp$, i.e. the support of o_s is empty. Otherwise, s is enabled and the support of o_s is not empty. However, if height$(o_s) < 1$, then their is no occurrence time for s, which is possible.

Before calculating the new fuzzy timestamp of the occurring situation, we need the relative fuzzy delay of the outgoing arc.

Definition 8.16 (Fuzzy Delay) *The* fuzzy delay $delay(t, p)$ *is the fuzzy time function associated with an arc* $(t, p) \in A$*, defining how long the token needs to arrive in* p *after firing from* t*.*

fuzzy delay

Now all elements needed for calculating the timestamp are available. If situation s occurs, we add to its occurrence time the fuzzy delay of the corresponding arcs.

Definition 8.17 (Fuzzy Timestamp) *The* fuzzy timestamp $\pi_{tp}(\tau)$ *of a token produced in situation* $s_t = (t, w_i, w_o)$ *is the possibility distribution of the time at which the token arrives in place* $p \in t\bullet$ *and is given by*

fuzzy timestamp

$$\begin{aligned}\pi_{tp}(\tau) &= o_t(\tau) \oplus delay(t, p)(\tau) \\ &= \begin{cases} \sup_{\tau=\tau_1+\tau_2} \min\{o_t(\tau_1), delay(t, p)(\tau_2)\} & \textit{if } \exists \tau_1, \tau_2 : \tau = \tau_1 + \tau_2 \\ 0 & \textit{otherwise} \end{cases}\end{aligned} \quad (8.19)$$

Remarks:
Eq. (8.19) is the unfolded version of the fuzzy addition (see (B.34)).

The firing in a FTVN removes and add tokens as usual, however, we do not define here, what to do if conflicts of enabled situations arise.

Definition 8.18 (Firing) *If a situation* $s_1 = (t_1, w_i, w_o)$ *occurs at marking* M_1*, the resulting marking* M_2 *is*

$$M_2 = (M_1 - w_i) + w_o \quad (8.20)$$

Finally, we can present a FTVN system, i.e. a FTVN net together with its dynamic semantics.

Definition 8.19 *A* FTVN system *is a six-tuple* $(N, \Sigma, C, delay, F, M_0)$ *where*

FTVN system

1. $(N, \Sigma, C, delay)$ *is a FTVN,*
2. F *is the firing function as defined above,*

3. *M_0 is the initial marking.*

If we have conflicts between quasi-enabled transitions, the timing information might resolve such conflicts if the enabling times are totally ordered. If the times are only partially ordered, i.e. the fuzzy times are overlapping, then the timing information does not resolve the conflict. In the literature, we found different approaches to deal with this situation:

1. Murata (1996) and Murata et al. (1999) consider all possible(!) firing sequences to determine the resulting timestamp at the final place. If there is a conflict, at least two subsequences exists, in which the timestamps (may) have different possibilities. The timestamps of each token in each place in all sequences are joined (i.e. $\cup$ is applied to them) resulting in the overall possibility distribution for the token's arrival time. In particular, this means that fuzzy timed Petri nets are not timed Petri nets in the usual sense, because the timing information is not used for resolving conflicts by priorities introduced by timing. To cite Murata (1996): "fuzzy timing makes partial orders not totally ordered, but adds some information on degrees of possibilities of event occurrence times".

2. Zhou and Murata (1999) define a new fuzzy measure, Ψ, as alternative to the well-known measures Π and $\mathcal{N}$ (see sec. B.1.6 on page 287), to define the order of firings for fuzzy timed occurrence times. Fuzzy measure Ψ makes partial orders totally ordered by applying a defuzzification method to the possibility distributions. But this reduces the set of possible time points to exactly one, which is not an appropriate interpretation of possibility distributions modeling vague concepts, such as a "short delay", which are used widely in Vitruv.

3. In a recent approach, Zhou et al. (2000) select a random realization of possibility distributions. This is used to simulate the timing behavior of the fuzzy-timing net.

The first approach is only feasible for small nets, in particular those where the occurrence graph is finite. The second approach is interesting, however, the reduction of a possibility distribution to a single value results in a deterministic timing, neglecting other possible occurrence sequences. Independent from the third approach, Marc Störzel and the present author (Störzel and Alfert, 2002) developed a simulation interpreting possibility distributions as scaled probability distributions (see sec. 12.1 on page 255), extending the simulation engine developed in Marc Stözel's diploma thesis (Störzel, 2001). Both approaches, from Zhou et al. (2000) and our, use possibility distributions as source for random variates and include explicitly possible timings in simulation runs.

In the following, we apply the simulation point of view, i.e. possibility distributions describe possible realizations of values, which can be analyzed by simulation runs.

8.2.2. Translating Interval Relationships of Vitruv_L to FTVN

With FTVN we can model static temporal relationships between intervals of Vitruv_L. In this section, we discuss the translation of the respective parts of Vitruv_L specifications to FTVNs. As in Vitruv_I, we do not model events, selectors and loops, their dynamic behavior is discussed later in sec. 8.4 on page 168. The composition of both, static and dynamic behavior is presented in sec. 8.6 on page 183. Therefore, it is sufficient to focus here on static temporal relationships only.

Since the structural facilities of Vitruv_L are not supported by FTVN and we do not model the event-based features of Vitruv_L, we can use the linearized specifications found in Vitruv_I. These linearized specifications have several advantages for the translation of interval relationships into a FTVN:

- the set of effective interval relations for each object is defined during the linearization of the Vitruv_L specification, we do not have to deal with the rules for nil-objects again (cf sec. 5.3.1 on page 64).
- if we need additional relations between intervals not explicitly given in the Vitruv_L specification, they can be inferred by the constraint solving algorithm on the basis of Vitruv_I.
- blocks define the maximal set of connected intervals without dynamic elements and thereby define the set of intervals in each FTVN.

The translation is driven by the block and scene structure, as defined for Vitruv_I. For each block exists a separate net. These nets are constructed independently, because no interval relations exists between intervals inside different blocks. However, any block in a scene can only be active, if the scene is active. Because of that, relationships exist between blocks inside a scene, they are discussed later in sec. 8.5.2 on page 181.

We identify for each interval, i, a transition, t_i, and a place, p_i, where $p_i \in t_i\bullet$. The delay, $delay(t_i, p_i)$, of the arc between t_i and p_i corresponds to duration of interval i. Consequently, activation of interval i is related to firing of transition t_i and producing a new token in p_i.

Interval relations between two intervals can be modeled as FTVN as shown in fig. 8.3 on the next page. In this figure, we show all 13 of Allen's interval relations, compare fig. 3.1 on page 31. We denote the delay of an arc as arc inscription @Da, where a corresponds to an interval (with transition t_a and place p_a), and Da is the interval's duration, i.e. $\text{Da} = delay(t_a, p_a)$. The notation is adopted from Jensen (1997). To obtain a connected net, it is sometimes necessary to introduce filler intervals between intervals a and b, e.g. if a before b, then there exists by definition a non-empty interval i, meeting b and met by a (this is possible because we work in $\mathcal{T} = \mathbb{R}_0^+$). These filler intervals are denoted with i and j and have durations Di and Dj, respectively. For easier comprehension, transitions, only needed for synchronization purposes between intervals, are drawn as black rectangles. Arcs starting at these transitions do

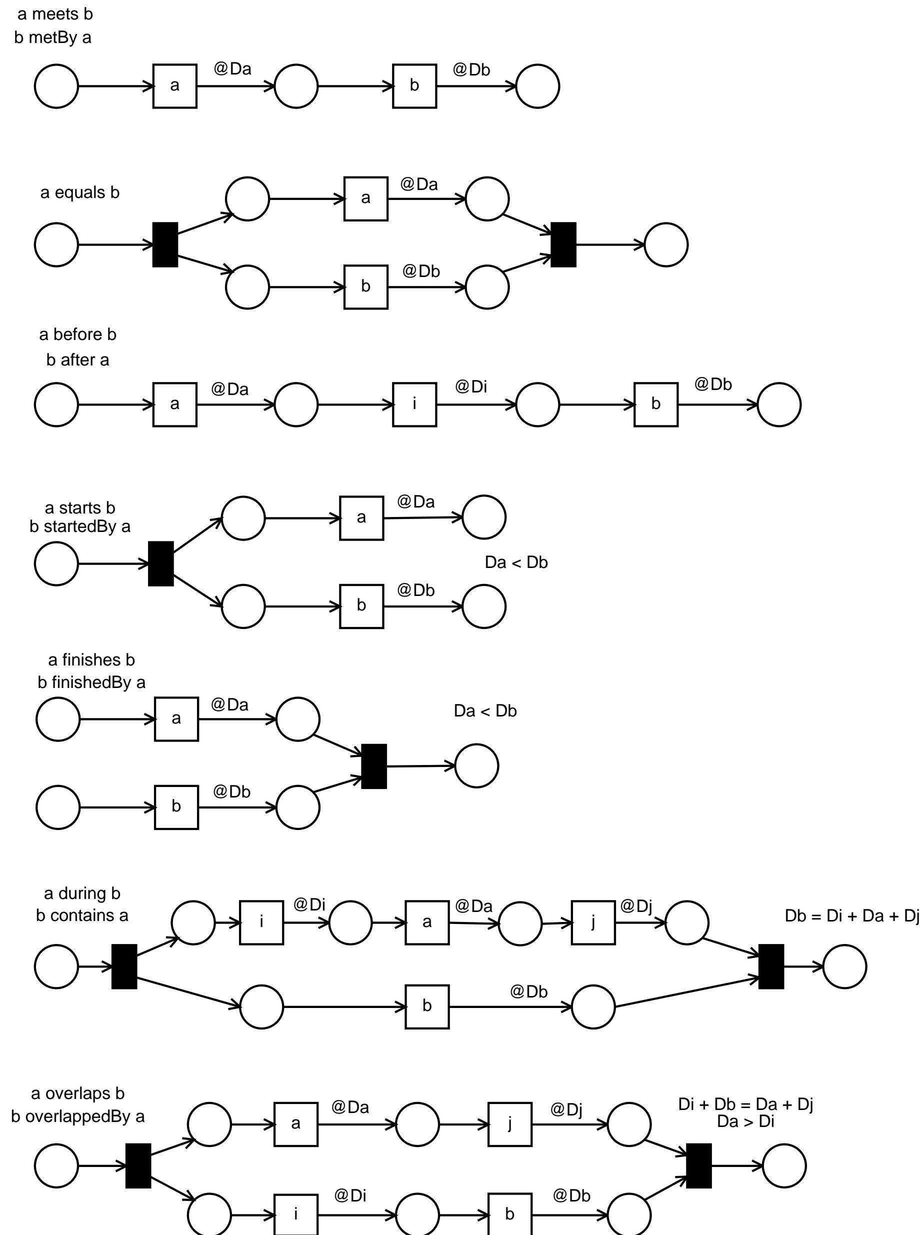

Figure 8.3.: Allen's Relations as Fuzzy Timing Vitruvian Nets. The black transitions fire without any time delay, all others have a fuzzy time delay.

not have delay inscriptions, because their delays have always a zero length: synchronization transitions fire without consuming time.

In fig. 8.4 on the next page we present the FTVN for class Example (first shown in spec. 5.2 on page 65), applying the FTVN representations of interval relations shown before in fig. 8.3 on the preceding page. We have four parallel paths in the net, according to the specification:

1. path alpha to omega of class Example,
2. path house1 to omega,
3. path transition to omega and
4. path house2 to omega,

The last three paths are overlapping each other, thus we need to introduce additional anonymous intervals for the path synchronization. The intervals are called i, j, k, l, m, and n, and are introduced by compound relations and some of the basic relations. We omit the fuzzy delays for reasons of simplicity, because it is clear that they occur on the post-set arcs of each transition representing an interval.

Additionally, we omit the sub-intervals alpha and omega, which by definition exist for each interval. However, for demonstration purposes, we show them for interval house2, they appear together with house2 in the dashed box. If no explicit relations between alpha, omega and the remaining intervals exist, – which is the case here except for alpha and omega of class Example – then sub-intervals alpha and omega can be omitted, because they do not change the overall behavior of the net.

Remarks:

We should note that we observe the same important properties of FTVN as Little and Ghafoor (1990) did for their OCPN nets, provided the net topology follows Allen's interval relations:

1. Apparently, the net is acyclic, because Allen's relations do not allow cyclic dependencies.
2. The net is also k-safe, which means that each marking has a at most k tokens at each place (Baumgarten, 1996, p. 131). This is because no place p has a preset of more than one transition, i.e. $\forall p \in P : |\bullet p| \leq 1$ and only one token is moved for each arc. Hence at most one transition puts a token into a place. For an initial marking with at most one token in each place, the net is 1-safe.
3. Because no cycle exists, after each initial marking M_0 exists a marking $M \in [M_0\rangle$, such that all transitions are dead, i.e. they cannot fire again. This marking M has only one token in the leaf place (cf. def B.34 on page 291). However, there is no deadlock: if we introduce a cycle from the leaf place to the root place which exactly one transition, the net is life: the final marking M_1 enables t, resulting in a token in the root place.

```
class Example
  exports house1 , house2 ;
  let
    house1 : video ;
    house2 : video ;
    transition : sound ;
  body
    alpha starts house1 ;
    house1 overlaps slightly transition ;
    transition overlaps slightly house2 ;
    house2 meets omega ;
  end ;
end ;
```

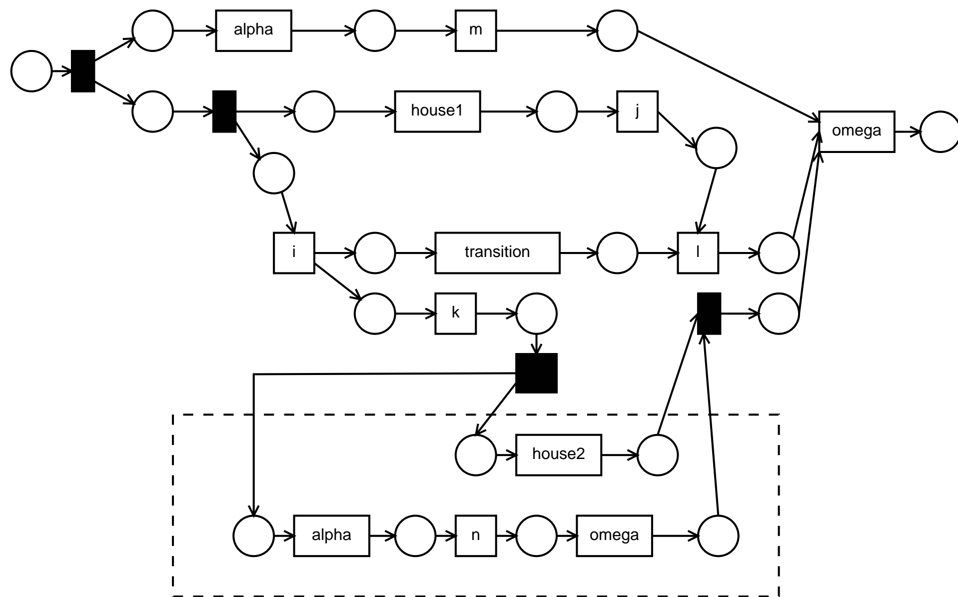

Figure 8.4.: The body of class Example as FTVN

8.3. Vitruvian Nets with Fuzzy Markings

Intervals and their relations are one important part of $\mathsf{Vitruv}_\mathsf{L}$. Another one deals with events and reactions to them. Event values are fuzzy sets, in selectors and loops we have expressions with these values selecting between different paths. Evaluation of fuzzy logic expressions is subject to Petri nets with fuzzy markings (also called fuzzy Petri nets), which we use to model the evaluation of event values in selectors and loops.

Fuzzy Petri nets have their origin in the works of Looney (1988) and are used for knowledge modeling and processing. They are generalized variants of elementary nets, in particular condition/events-nets, where Boolean valued tokens are replaced by tokens representing fuzzy values (Fay and Schnieder, 1999). The cited approaches use only fuzzy truth values and apply the net dynamics to maintain continuously truth values in rule based knowledge management systems.

In contrast to the literature, we use fuzzy Petri nets to non-continuously evaluate fuzzy logic expressions in a fuzzy rule based system, i.e. we evaluate these expressions only at particular, discrete time-points. First, we present an example how to model a selector, after that we give the formal definition.

8.3.1. Translating Selections in $\mathsf{Vitruv}_\mathsf{L}$ to Fuzzy Vitruvian Nets

We use Fuzzy Vitruvian Nets (FVN) to model the selection process in selectors and loops. After an event occurs, we have to decide which path in selectors and loops control will take depending on the event's value. This decision process can easily be modeled with a FVN.

In fig. 8.5 on the next page we show an example for a selection between two alternatives. The corresponding code fragment above the net shows a selector, s, with two branches. The first is chosen, if the event value is compatible to f1, the second branch if the event value is compatible to f2.

The FVN represents the selection process, therefore neither the event nor the bodies of the branches are of interest here. The event value is realized as token in place root; the token value is the value of the event. The comparison between event value and the two constants f1 and f2 is done in the two compat-transitions, therefore we copy the token in place root to the pre-set of both compat-transitions. The gen-transitions produce new tokens with fuzzy set values f1 and f2, resp., required for the tests of compatibility. The gen-transitions have no pre-set, therefore they can fire until the maximum capacity of their post-set is reached. We restrict all places in the FVN to a capacity of 1 to ensure a sequential and stepwise firing process.

The tests for compatibility of event value v and constant value f in the compat-transitions result in fuzzy truth values denoting the degree of compatibility, which is computed by function *simil* (defined in eq. (B.49)). Constant f is the pattern value to be matched by event value v, i.e. we apply $\mathit{simil}(\mathsf{v}, \mathsf{f})$. To encode which the tests

```
s selects (..) with rules
  on f1 do ... end;
  on f2 do ... end;
end;
```

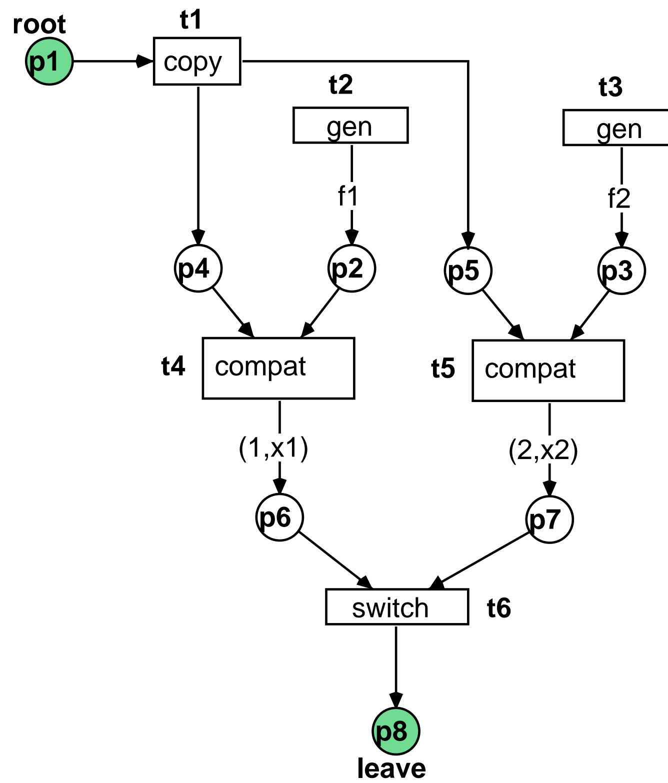

Figure 8.5.: Comparison of a value with two alternatives.

yields which result value, we enumerate the tests. The token produced by the compat-transitions is pair of the ordinal number of the test and the comparison result: (1, x1) and (2, x2), for the comparison with f1 and f2, resp., where x1 and x2 hold the respective degrees of compatibility.

The decision of the highest compatibility is carried out by transition switch, resulting in a token with a natural number value in place leave. In this example, the possible value in leave is either 1, 2 or 0, depending whether the value of the initial token in root, i.e. the event value, is compatible to f1, f2 or neither.

The construction used in fig. 8.5 on the facing page can easily be extended to more than two alternatives, because we have only to add further tests of compatibility together with their generators of the test values, additional copies of the token in place root, and connections between the result of the tests and transition switch.

The construction can also be reduced to check the termination condition of a loop. We have only to remove one of the two tests, together with its test value generator and its copy transition of the token in place root. We have then only one path from root to leave and hence only one condition to be checked.

8.3.2. Formal Definition

Fuzzy Vitruvian Nets (FVN), constructed as shown in fig. 8.5 on the preceding page, extend abstract Vitruvian Nets by constraining Σ and introducing new components. Each FVN models the decision procedure of one event, therefore all fuzzy sets in a FVN have the same universe X of the event value. Transitions can be partitioned into four groups: copying tokens; generating new tokens used as constant values in the comparison; test for compatibility; and finally selecting which of the tests was most successful. We can assign a symbol to each transition, denoting the partition to which it belongs. Additionally, we need the constant values generated by gen-transitions and a mapping between transition and value.

Definition 8.20 (FVN) *A* FVN *is a seven-tuple* $(N, \Sigma, C, X, Sym, f, g)$ *where*

FVN

1. (N, Σ, C) *is an abstract VN,*
2. X *is a universe for fuzzy sets,*
3. $\Sigma = \{\mathcal{F}(X), [0,1], \mathbb{N}, \mathbb{N} \times [0,1]\}$
4. *Sym is a set of function symbols,* $Sym = \{copy, gen, compat, switch\}$*,*
5. $f = T \to S$ *assigns a function symbol to each transition,*
6. $g : T' \to \mathcal{F}(X)$*, where* $T' = \{t \mid t \in T \wedge f(t) = gen\}$*, assigns a constant fuzzy set to transitions with function symbol gen,*
7. *the arc relation* A *of* N *is acyclic,*

8. *the capacity of each place $p \in P$ is* 1.

The coloring of places depends on the assigned function symbols of the associated transitions, as shown in table 8.1. Function symbol gen denotes generator transitions, which have no pre-set.

$f(t)$	$C(p), p \in \bullet t$	$C(p), p \in t\bullet$
copy	$\mathcal{F}(X)$	$\mathcal{F}(X)$
gen		$\mathcal{F}(X)$
compat	$\mathcal{F}(X)$	$\mathbb{N} \times [0, 1]$
switch	$\mathbb{N} \times [0, 1]$	$\mathbb{N}$

Table 8.1.: Token colors and transition function symbols

$\mathcal{F}$

The set of all FVN is denoted with $\mathcal{F}$.

Remarks:

1. Fuzzy Vitruvian Nets are rather restricted, because we need them only for modeling the path selection in selectors and the termination condition for loops. In particular, it is impossible to evaluate general fuzzy logic expressions, however it is simple to extend the FVN definition for modifiers, union and intersection, etc., together with a suitable firing function.
2. X is a generic parameter resulting in $\mathcal{F}(X) \in \Sigma$, all other elements of Σ are independent of the type of event value.
3. g is the mapping between generator transitions and their respective fuzzy set, produced by the transitions.
4. In fig. 8.5 on page 164 we have seen, that the net is acyclic, we enforce it here in the definition.

Now we define the dynamic semantics of FVN, i.e. the interpretation of the function symbols, the firing function F, and the enabling of transitions.

In switch transitions, we decide which of the compatibility tests has the highest degree of compatibility. For this decision, we define function m. If all parameters have a compatibility degree of 0, m returns 0. If we have only one parameter, we apply a threshold value as minimal required degree of compatibility.

Definition 8.21 *Partial function* $m : (\mathbb{N}^+ \times [0, 1])^n \nrightarrow \mathbb{N}, n \geq 1$, *is defined as follows for* $n > 1$*:*

$$m((l_1, x_1), \ldots, (l_n, x_n)) = \begin{cases} 0 & \text{if } 0 = \max\{x_1, \ldots, x_n\} \\ l_k & \text{if } x_k = \max\{x_1, \ldots, x_n\}, 1 < k < n \end{cases} \tag{8.21}$$

If several maxima exist, a nondeterministic choice is made between them. If we have only one parameter (l, x)*, we apply threshold* $t = 0.5$ *as a measure of a required minimal compatibility:*

$$m(l, x) = \begin{cases} 0 & \text{if } x < t \\ l & \text{otherwise} \end{cases} \tag{8.22}$$

For parameters (l_k, x_k)*, we demand that all* l_k *are positive and pairwise not equal.*

A situation, $s = (t, w_i, w_o) \in \mathbb{S}$, is enabled if enough tokens in the pre-set of t are available and the capacity constraint for the post-set of t is satisfied: transition t is only allowed to fire if its post-set, $t\bullet$, is empty.

Definition 8.22 (Enabling) *Let* $(N, \Sigma, C, X, Sym, f, g)$ *be a FVN. Transition* $t \in T$ *is* enabled *in situation* $(t, w_i, w_o) \in \mathbb{S}$ *and marking* M *if*

enabled

$$w_i \subseteq_{ms} M \tag{8.23}$$

$$\forall p \in t\bullet : |M(p)| = 0. \tag{8.24}$$

The firing behavior of a transition in a FVN depends on its assigned function symbol: either copying tokens, generating tokens, comparing tokens or switching. Therefore, we can define explicitly the firing function for transitions as function expressions, depending on the function symbol, i.e. we can define how to calculate w_o for transition t and for w_i such that $F(t)(w_i) = w_o$.

Definition 8.23 (Firing Function) *The firing function, F, of FVN* $(N, \Sigma, C, X, Sym, f, g)$ *for transition t depends on the function symbol* $f(t)$*. Let* $\{p_{i1}, \ldots, p_{ik}\} = \bullet t$ *and* $\{p_{o1}, \ldots, p_{ol}\} = t\bullet$ *be ordered in some arbitrary, but fixed way. Let* $v, v_i \in C(p)$ *where* $p \in nb(t)$*. The firing function, F, is defined for each transition t as*

$$F(t) = \begin{cases} \{(p_{i1}, v)\}_{ms} \mapsto \{(p_{o1}, v), \ldots, (p_{ol}, v)\}_{ms} & \text{if } f(t) = copy \\ \emptyset \mapsto \{(p_{o1}, g(t)), \ldots, (p_{ol}, g(t))\}_{ms} & \text{if } f(t) = gen \\ \{(p_{i1}, v_1), (p_{i2}, v_2)\}_{ms} \mapsto \{(p_{o1}, (k, simil(v_1, v_2)))\}_{ms} & \text{if } f(t) = compat \\ \{(p_{i1}, v_1), \ldots, (p_{ik}, v_k)\}_{ms} \mapsto \{(p_{o1}, m(v_1, \ldots, v_k))\}_{ms} & \text{if } f(t) = switch \end{cases} \tag{8.25}$$

where simil is the similarity measure for fuzzy sets (cf. (B.49)*).*

Remarks:

1. Copy transitions copy the token in the pre-set to all places in the post-set.
2. Each generator transition, t, has no pre-set, but put its defined token value, $g(t)$, into all places of its post-set.
3. Comparison transitions apply function *simil* as measure for the degree of compatibility between v_1 and v_2, where v_2 is the reference value. Value $k \in \mathbb{N}^+$ is a constant for each transition, and is used to differentiate comparison transitions, if their post-set is in the pre-set of the same switch transition.

4. Switch transitions apply function m, hence token values of the pre-set are pairs $v_i = (l_i, x_i) \in \mathbb{N} \times [0, 1]$. Function m requires that all l_i are pairwise not equal.

Example 8.2 Firing function, F, for the net in fig. 8.5 on page 164 is:

$$
\begin{aligned}
F = \{ & t_1 \mapsto \{\{(p_1, v_1)\} \mapsto \{(p4, v_1), (p_5, v_1)\}\} \\
& t_2 \mapsto \{\emptyset \mapsto \{(p_4, f_1)\}\} \\
& t_3 \mapsto \{\emptyset \mapsto \{(p_5, f_2)\}\} \\
& t_4 \mapsto \{\{(p_4, v_4), (p_2, v_2)\} \mapsto \{(p_6, (1, \mathit{simil}(v_2, v_4)))\}\} \\
& t_5 \mapsto \{\{(p_5, v_5), (p_3, v_3)\} \mapsto \{(p_7, (2, \mathit{simil}(v_3, v_5)))\}\} \\
& t_6 \mapsto \{(p_6, v_6), (p_7, v_7)\} \mapsto \{(p_8, m(v_6, v_7))\}\} \\
\}
\end{aligned}
$$

□

As usual, a FVN system combines a net together with the firing function and an initial marking.

FVN system

Definition 8.24 (FVN System) *The nine-tuple* $(N, \Sigma, C, X, Sym, f, g, F, M_0)$ *is a* FVN system *where*

1. $(N, \Sigma, C, X, Sym, f, g)$ *is a FVN,*
2. F *is the firing function as defined above,*
3. M_0 *is the initial marking.*

8.4. Nets for Events, Selectors and Loops

To translate a complete Vitruv_L specification, we combine FTVNs for blocks and FVNs for decisions in a more general net class, the basic Vitruvian Net (sec. 8.4.1 on the next page). With them we can model events, selectors and loops.

Selectors and loops are the glue between blocks of a scene. Both depend on events. If an event occurs, its value is fed into a decision procedure either for a selector, choosing an alternative path, or for a loop, deciding about the loop's termination. The decision procedures are modeled by FVNs. We will discuss how to combine FTVNs for blocks and FVNs for decision procedures with events for selectors and loops, respectively. Our strategy is to use particular net structures for events (sec. 8.4.2 on page 172), selectors (sec. 8.4.3 on page 174) and loops (sec. 8.4.4 on page 176), respectively. Because these net structures are never used alone, we call these structures subnets. They take part in larger nets.

8.4.1. Basic Vitruvian Nets

Modeling events, selectors and loops requires combining both, FTVN, for modeling bodies, and FVN, for decision procedures selecting between different bodies. Therefore, our new net class shares characteristics of both, FTVN and FVN, and thereby of abstract Vitruvian Nets. But we need some more properties.

The main characteristics of events are that they generate random values at a random time, modeling the user's choice. In a Petri net, we can model this by transitions producing tokens with random values and random delays. In particular, the delays might be different each time the transitions fires.

If we have several events, they might have different universes for their values. Hence we cannot use one set X of FVN as the single universe. Therefore, we generalize X to a family of universes (X_j) and extend the set of token types by $\mathcal{F}(X)$for each $X \in X_j$.

Modeling of selectors and loops requires complex synchronizations between transitions. Therefore, we also add as structural elements inhibitor and reset arcs (see def. B.38 on page 292 and B.37 on page 292). Their use is shown in the particular nets for selectors and loops in sec. 8.4.3 on page 174 and sec. 8.4.4 on page 176, respectively.

Basic Vitruvian Net

Definition 8.25 (Basic Vitruvian Net) *A* Basic Vitruvian Net *(BVN) is a ten-tuple* $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R)$ *where*

1. $N = (T, P, A)$ *is a Petri net,*
2. $(X_j)_{j \in J}$ *is a family of universes for* $j \in J$, $J = \{j_1, \ldots, j_n\}$ *is an index set,* $\mathcal{T}$ *is an element of* (X_j),
3. Σ *is a set of token types (color sets) including* $\{\star\}, \mathbb{N}, [0,1], D, \mathcal{F}(X_{j_1}), \mathcal{F}(X_{j_2}), \ldots, \mathcal{F}(X_{j_n})$ *and (n-ary) product types with* $\sigma_1 \times \ldots \times \sigma_n \in \Sigma \Rightarrow \sigma_1, \ldots, \sigma_n \in \Sigma$,
4. $C : P \rightarrow \Sigma$ *assigns to each place a type,*
5. $delay : T \times P \nrightarrow \mathcal{D}$ *assigns fuzzy delays to arcs from transitions to places,*
6. *Sym is a set of function symbols,*
7. $f : T \nrightarrow S$ *assigns a function symbol to transitions,*
8. $g : T' \rightarrow F$, *where* $T' = \{t \mid t \in T \wedge t \in \mathrm{dom}(f) \wedge f(t) = gen\}$, *assigns a constant fuzzy set to transitions with function symbol gen.*
9. $I \subseteq P \times T$ *are inhibitor arcs,*
10. $R \subseteq P \times T$ *are reset arcs.*

$\mathcal{B}$

The set of all BVN is denoted by $\mathcal{B}$.

Remarks:

1. The mentioned types $\{\star\}, \mathbb{N}, [0,1], D, \mathcal{F}(X_{j_1}), \ldots, \mathcal{F}(X_{j_n})$ are basic types. We can build recursively more complex types as Cartesian products of elements of Σ. This is needed for timed color sets, such as $\{\star\} \times D$, which is the (only) color set in FTVN.
2. The set of fuzzy durations, D, is included in $\mathcal{F}(X_{j_i})$, because $\mathcal{T} \in X_j$, however, for fuzzy timing of transitions, we need D explicitly.
3. Function f is in contrast to the definition in FVN only a partial function, because not all transitions are defined in the same way as in FVN.
4. A FTVN can be expressed as a BVN, where I, R, f and g are empty sets.
5. A FVN can be expressed as a BVN, where I and R are empty sets, all delays are zero, i.e. $\forall (t,p) \in A : \mathit{delay}(t,p) = d_0$, and f is a total function ($T = \mathrm{dom}(f)$).

The markings of a BVN are more complex than in FTVN, because the color sets are more complex. Therefore we refine both, timed and untimed markings. In contrast to FTVN, we do not necessarily have tokens with a fuzzy duration as its value or part of its value. A marking, where at least one token has a fuzzy duration as the last component of its value (this is compatible to the definition in FTVN), is called a timed marking, otherwise it is called an untimed marking.

timed marking

Definition 8.26 *Let $M \in \mathcal{M}(W)$ be a marking of BVN. M is a* timed marking *if and only if*

$$\exists (p,v) \in_{ms} M : C(p) = \sigma \times \mathcal{D} \tag{8.26}$$

untimed marking

timed color set

otherwise M is called an untimed marking*. We call σ_1 a* timed color set *with $\sigma_1 = \sigma \times \mathcal{D}$ and $\sigma_1, \sigma, \mathcal{D} \in \Sigma$.*

Remarks:

1. For a timed marking, we need a color set of at least one place which is a product of at least two color sets, because the duration is a kind of attachment to another possible token value. Token type $\sigma \times \mathcal{D}$ is the general form of such product types of a timed marking, where σ matches any type in Σ, including product types.
2. In a FTVN, σ is always $\{\star\}$, and the color set of each place is the pair $\{\star\} \times D$.
3. If $\forall (p,v) \in_{ms} M : C(p) = \mathcal{D}$, then M is an untimed marking, although the token values are durations. For timed markings, however, we require tokens which can exist with their fuzzy timestamp stripped. This would be here not the case, if we strip $\mathcal{D}$ from $C(p)$ type, the remaining color set would be empty.

With the notion of timed color sets we can define an additional constraint for *delay*: its domain contains all arcs to places, which have a timed color set as type. This preserves the structure constraint of *delay* defined for FTVNs.

Definition 8.27 *Let B be a basic Vitruvian Net. The domain of delay contains all arcs to places with a timed color set:*

$$\mathrm{dom}(delay) \supseteq \{(t,p) \mid (t,p) \in A \wedge C(p) = \sigma \times D\} \quad \textit{for some } \sigma \in \Sigma \tag{8.27}$$

For defining quasi-enabled situations, we need an untimed version of a timed marking. Therefore, we define the untimed marking of a timed marking similar to FTVN by dropping the last component of the token value, if the last component is a fuzzy duration. Again, we require that the token value with the fuzzy duration has at least two components. We generalize function pr_{FTVN} of def. 8.11 on page 154 to deal with the more complex color sets, the remainder of the definition is taken from def. 8.11.

Definition 8.28 *Let $M \in \mathcal{M}(W)$ be a timed marking of BVN. The untimed marking, M^u, of timed marking M is defined as the formal sum of all untimed elements of M:*

M^u

$$M^u = \sum_{m \in M} \{\mathrm{pr}(m)\} \tag{8.28}$$

where function pr *is defined as*

pr

$$\mathrm{pr} : ((P \times \Sigma) \times \mathbb{N}) \to ((P \times \Sigma) \times \mathbb{N}) \tag{8.29}$$

$$\mathrm{pr}((p,\sigma),n) = \begin{cases} ((p,\sigma'),n) & \textit{if } \sigma = \sigma' \times \mathcal{D} \\ ((p,\sigma),n) & \textit{otherwise} \end{cases} \tag{8.30}$$

Remarks:

1. If $C(p)$ is timed color set (i.e. $C(p) = \sigma' \times \mathcal{D}$), function pr strips the fuzzy timestamp, otherwise pr is the identity function.
2. The un-timing operator $\cdot^u$ is not idempotent, i.e. $(M^u)^u \neq M^u$: consider marking $M = \{(p,(\star,d_1,d_2))\}_{ms}$, where $d_1, d_2 \in D$. $M^u = \{(p,(\star,d_1))\}_{ms}$ is again a timed marking, and hence $(M^u)^u = \{(p,\star)\}_{ms}$.

As we have timed markings, we need the definition of quasi-enabled situations. However, we have to refine the definition of FTVN, because we have inhibitor arcs for further constraints of the enabling: if transition t is quasi-enabled, then all places of inhibitor arcs of transition t need to be empty.

Definition 8.29 (quasi-enabled) *Situation $s = (t, w_i, w_o)$ is* quasi-enabled *in marking M if*

quasi-enabled

$$w_i^u \subseteq_{ms} M^u \tag{8.31}$$

$$\forall p \in P : (p,t) \in I \Rightarrow M(p) = 0. \tag{8.32}$$

The firing behavior is extended by reset arcs. We will not mention them in the definition of firing function F for each transition t, because connections between places and transitions by reset arcs are clear from relation R (or equivalently from its graphical representation). Thus, we present here the rule for calculating the new marking when a transition fires, depending on both, function F and the reset arcs.

Definition 8.30 (Firing behavior) *Let F be the firing function of a BVN, and $s = (t, w_i, w_o)$ a quasi-enabled situation in marking M_1. If s occurs, the new marking M_2 is*

$$M_2 = M_3 + w_o \tag{8.33}$$

where

$$M_3 = M_4 - \{M_4(p) \mid p \in P \wedge (p, t) \in R\}_{ms} \tag{8.34}$$
$$M_4 = M_1 - w_i \tag{8.35}$$

Remarks:

1. If M_4 cannot calculated because $w_i \not\subseteq M_1$, then we cannot fire. This shall, however, never happen, because in such a case the situation $s = (t, w_i, w_o)$ is not quasi-enabled.
2. M_4 is the marking of net after w_i is removed from M_1. For M_3, we remove all (remaining) tokens in all places connected by a reset arc to t from M_4.

The remaining definitions of the firing behavior of both, FTVN and FVN, are transferred to BVN, such that their behavior is preserved. Calculating fuzzy timestamps and applying the firing functions for the four functions symbols in *Sym* are defined for BVN in the same way as in the original nets. With these definitions we can define basic Vitruvian Systems combining, as before, the net and its behavior.

Basic Vitruvian System

Definition 8.31 (Basic Vitruvian System) *A* Basic Vitruvian System *is a twelve-tuple $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, F, M_0)$, where*

1. *$(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R)$ is a BVN,*
2. *F is the firing function, and*
3. *M_0 is the initial marking.*

8.4.2. Event Subnets

For each event, we have to consider that during its enabling time the event may occur multiple times. For loops it is important to have access to all occurring event values,

for selectors we are only interested into the first value. Additionally, we need the information whether no event occurred during the enabling time, resulting in the TimeOut value. All these considerations can be found in the net shown in fig. 8.6 on the following page.

We discuss different paths through the net. The occurrence of the event is modeled by transitions t3 and e.oc, the former fires with a random delay, the latter fires a random value of the event's type. For simulation, appropriate probability distributions may be attached to both transitions to describe properly the application-specific random behavior. However, discussing which distributions are appropriate is beyond the scope of this thesis and thus is not discussed further.

Transition e.oc has three elements in its post-set:

1. the event's value is put in place Value;
2. one token is moved back into the enabling place p1 of e.oc allowing multiple occurrences;
3. one token is put in a place in the pre-set of transition hasFired, indicating that the event occurred and thus no TimeOut value will be generated.

To prevent e.oc from firing after the enabling of the event has finished, we have an inhibitor arc from place End to transition e.oc with an hollow circle instead of the arrow of ordinary arcs.

The enabling interval of an event is modeled by transition e.en and its following place together with fuzzy delay Den. This is the only fuzzy timed transition in the entire event subnet. Immediately after the enabling time of the event we have two possible situations:

1. the event occurred, then a token is put in the post-set of transition hasFired, enabling transition t1, adding a token into place End and additionally removing all tokens from p2;
2. the event has not occurred, transition t2 can fire, removing tokens from p1 and p3, and adding tokens to End and TimeOut.

Parallel to the start of the event enabling, one token is put in place Enabled to indicate that the event is currently enabled. It is used by selectors (sec. 8.4.3 on the next page).

This net will be used later for all events, because they all share the structure and the behavior. The net is a kind of a parameterized macro, similar to machine code macros for certain high-level instructions found in code generators of compilers. As an abstraction, we fold the net to a transition with pre-set Start and the post-set Value, Enabled, End and TimeOut as indicated by the gray rectangle. This allows also easier identification of event subnets in larger nets. The parameters of the net for events are

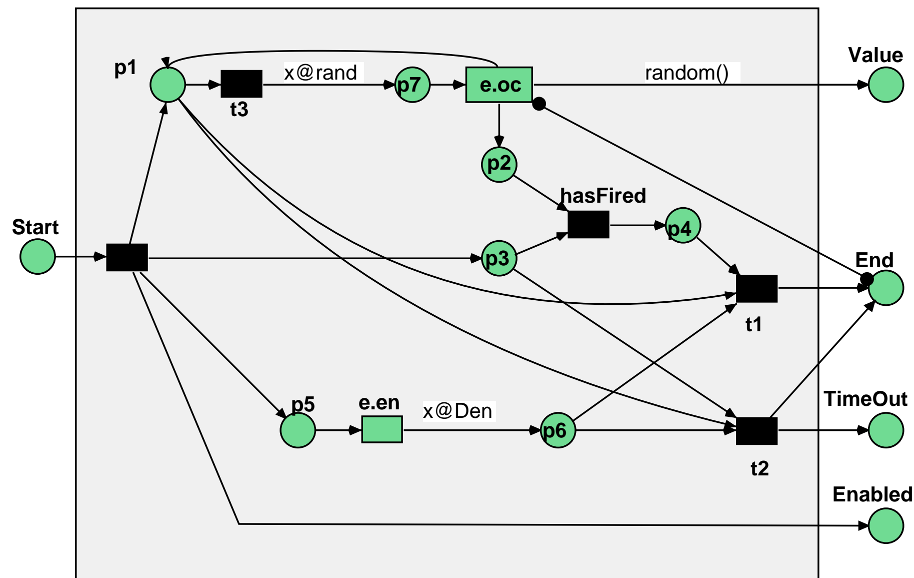

Figure 8.6.: BVN for Events.

- the type of place Value, corresponding to the type parameter of the event declaration in Vitruv_L,
- the enabling time, in the net denoted as @Den,
- probability distributions for both, the generated values and their occurrence delays; they are used for simulation purposes.

If we have ***multiple reactions*** to an event, we additionally need for each output place (i.e. Value, End, TimeOut, and Enabled) a one-to-many copy-transition, which copies the token from the one pre-set place to its many post-set places. In such a case, a selector or loop reacting on the event, is not connected directly with the output places of the event, but rather with post-set places of the respective copy-transitions. With this construction, we can easily support an unlimited (but finite) number of reactions to the same event. However, for reasons of simplicity, we do not show these copy transitions in the applications of event subnets in the following sections.

8.4.3. Selector Subnets

Nets for selectors apply both, an event subnet and a FVN for decisions. The bodies of the alternative paths are Basic Vitruvian Nets, because they may contain again selectors and loops. The selector net is shown in fig. 8.7 on the facing page and is explained next.

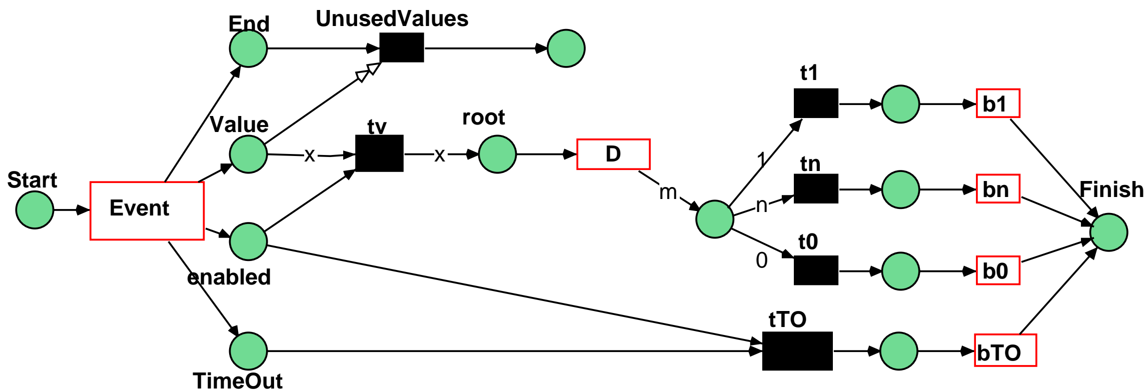

Figure 8.7.: The net for selectors.

The overall structure of a selector net is divided into a pipeline of three parts:

1. the selector's event net, here shown only as a single white transition with inscription Event. The value of the event is fed into
2. the decision procedure, a FVN, shown as single white transition with inscription D. Either its result, the number of the selected alternative, or the token in place TimeOut triggers one of
3. the alternative bodies, shown as white transitions with inscriptions b1, bn and bTO.

Between these different parts we need several places and transitions defining details of synchronization and control aspects. We discuss them now, moving from left to right.

In contrast to loops, selectors react only on the first value generated by their event. Therefore, transition tv fires only if the event is enabled (token in place enabled) and a value is available in place Value. After firing of tv, the value-token is fed into transition D. Since the only token in enabled is removed, transition tv cannot fire again, until the event is re-enabled, preventing multiple activation of selector alternatives during the event's enabling interval. If the enabling interval is finished, the event net puts a token in TimeOut. We can distinguish two cases:

1. if the event has occurred, we might have some unused values in place Value and place enabled is empty. Transition UnusedValues can fire, removing all tokens left in Value via the reset arc between Value and UnusedValues. The reset arc is notated with a double hollow triangle instead of the arrow of ordinary arcs.

2. if the event has not occurred, we have no values in place Value, but the token in enabled is still there. Now transition tTO fires activating the selector's alternative for the TimeOut-value.

In the FVN, denoted with transition D, the decision for selecting an alternative body has to be made. Its result in place leave is an integer value $m \in [0, n]$ if we have n alternative bodies. Value 1 activates the first body, 2 the second, and so on. Value 0 denotes that the event value lacks compatibility to any test values, hence no body is activated. The activation is controlled via the arc inscriptions indicating the corresponding values, i.e. the firing function F for transition tm with $m \in [0, n]$ is

$$F(\mathsf{tm})(\{(\mathsf{leave}, m)\}_{ms}) = \begin{cases} \{(\mathsf{pm}, \bigstar)\}_{ms} & m > 0 \\ \{(\mathsf{leave}, \bigstar)\}_{ms} & m = 0 \end{cases}$$

After the event is activated, either one of the n bodies is activated, depending on the event value and the decision procedure D, or body bTO is activated by the TimeOut-event. Finally, a token is put into Finish, indicating that the selector is finished.

For a proper application of the selector net, we have to satisfy the following conditions for glueing together the various nets:

- the type of place Value, i.e. $C(\mathsf{Value})$, has to be the same as of place root ($C(\mathsf{root})$): event and decision procedure match.
- the number of different selector alternatives b1, . . . , bn has to match the number of branches inside D, resulting in the proper interval range in place leave. This means that the number of antecedents equals the number of consequents, taken from the corresponding selector statement in $\mathsf{Vitruv_L}$ but excluding the TimeOut branch.

8.4.4. Loop Subnets

Nets for loops apply both, an event subnet and a FVN for decisions. The loop body is a Basic Vitruvian Net because it may contain events, selectors and loops. The structure of the net is more complex than that of the selector net, because we have to deal with multiple event values during event enabling interval, and true concurrency between the loop body and the occurring event. The net is shown in fig. 8.8 on the next page and explained now.

A loop net consists three different parts:

1. the loop's event net, shown as a single white transition with inscription Event. The value of the event is fed into
2. the decision procedure, a FVN shown as a single white transition with inscription D. Its result, either 0 or 1 triggers execution of

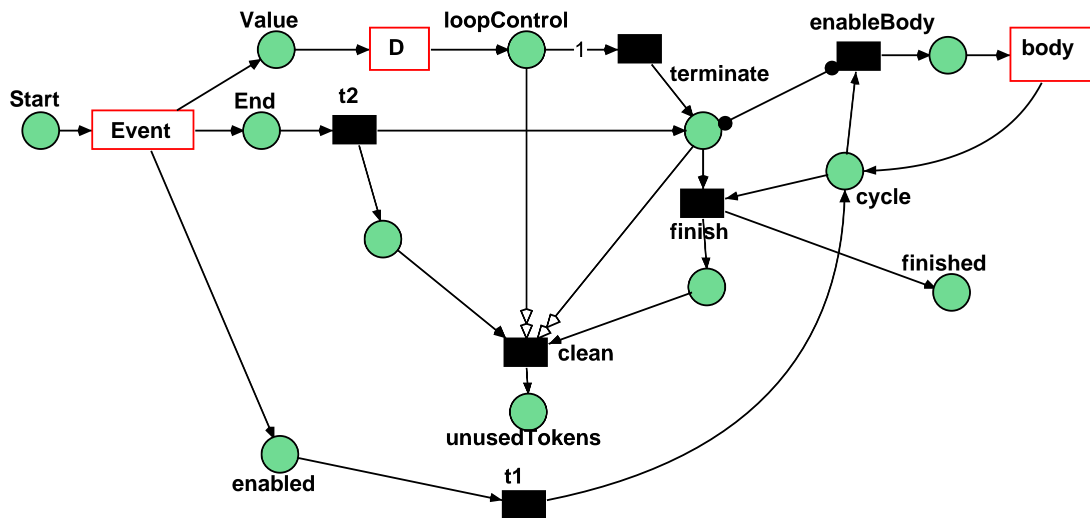

Figure 8.8.: The net for loops.

3. the loop body, shown as a single white transition with inscription body.

Between these parts several places and transitions exist controlling the synchronization between them. We discuss now the structure in detail, moving from left to right.

In contrast to selectors, loops depend not only on the first event value: the loop body remains active until the termination condition is true or the event enabling is finished. Thus, we feed all event values (place Value) immediately into the decision net D, i.e. place Value and D's place root are merged. In place loopControl we collect the results of D, i.e. loopControl and D's place leave are also merged. In loopControl the token values are either 0 or 1, where 0 means that the termination condition is not met, 1 means the opposite.

The loop body can be activated immediately after the event is enabled. Therefore, transition t1 occurs immediately after a token appears in place enabled, indicating that the event is enabled. Transition t1 creates a token in place cycle, enabling the loop body. Now the loop can start after firing of transition enableBody, provided no event satisfying the termination condition has occurred in parallel, since this would result in a token in place terminate. Any token in place terminate disables transition enableBody because of the inhibitor between terminate and enableBody.

If the termination condition is satisfied, a token is in place terminate. After the current activation of the loop body, i.e. after place cylce obtains a token, transition finish fires, removing the token from cycle and creating a new one in finished. Now the loop cannot be activated until the event is re-enabled: the loop is finished and intervals following the loop body can be activated.

An additional final step has to be done after the event's enabling interval is finished,

i.e. a token is created in place End. Transition t2 fires creating a token in terminate. This token terminates the loop even in the case that all other event values have not satisfied the termination condition. Transition t2 creates also a token in the pre-set of clean, which becomes enabled after finish has fired. Transition clean has reset arcs to places loopControl and terminate, i.e. after firing of clean the net's places are empty except for unusedTokens and finished.

The following conditions have to be satisfied for properly glueing together the subnets for the loop net:

- the type of place Value (C(Value)) has to be the same as of place root (C(root)) of the FVN D, i.e. event and decision procedure match.
- we have only one condition in D resulting in values 0 or 1 in leave, which is the same place as loopControl.

8.5. Nets for Scenes

Scenes are the largest constructions in Vitruv_L. Combining a set of blocks connected by events, selectors and loops, scenes are also the largest nets in the dynamic semantics for a Vitruv_L specification. We will now discuss the structure of scenes and their connection by jumps (sec. 8.5.2 on page 181) and present an algorithm to construct the entire net of a presentation (sec. 8.6 on page 183). Before that we introduce Vitruvian Nets, an extension of BVN in the next section.

8.5.1. Vitruvian Nets

Scenes combine blocks, events, selectors and loops. These ingredients can be modeled by basic Vitruvian Nets, as we have seen in the sections before. Most important now is that scenes are simply objects and thus intervals. Intervals have a definitive start and end point, resp., all other activities inside the scene occur between these two points. Therefore, we can model scenes as a FTVN, as we have done for ordinary intervals, but augmented by events, selectors and loops. That is why we need BVNs here.

Leaving the current scene, let us say s, for a new scene, s', requires removing all tokens from all places of s. This can easily be modeled by reset arcs, however, the selection of all places of s is rather tedious, because we have to infer them from the interval relations. Therefore, we introduce scenes as an explicit structural element of Vitruvian Nets, realized as a set, Sc, of scenes and map, S, assigning to each place a scene. Then it is easy to determine the set of places of scene s, for which reset arcs are needed if we leave scene s.

The very first scene of presentation is defined in Vitruv_L as the first scene is in the binding. The class specification does not specify which of the scenes starts the presentation. To reflect this using Vitruvian Nets, it is useful to include Sc in the set of

color sets, Σ. A place, p, with color set Sc connected to the first transition of all scenes can be used to trigger the start of any scene s, if the token value of p is the scene $s \in Sc$ to be started. This makes the net topology independent of the order of scenes in the binding, because only the token value of p defines which scene is started first. However, this place p is not a part of any scene defined in the $\mathsf{Vitruv_L}$ specification, and therefore we cannot assign a unique scene to p, any scene may be as possible. This violates, however, the structure of $\mathsf{Vitruv_L}$, where anything can uniquely be assigned to a scene. Therefore, we introduce a distinguished element $sc_0 \in Sc$, denoting a scene which is not a regular presentation but rather an artifical scene, i.e. it has no counterpart in the $\mathsf{Vitruv_L}$ specification. Accordingly, sc_0 is the assigned scene of the aforementioned place p, residing outside regular scenes defined in $\mathsf{Vitruv_L}$.

Vitruvian Nets are an extension of Basic Vitruvian Nets, hence they inherit features from Fuzzy Timed Vitruvian Nets and from Fuzzy Vitruvian Nets.

Vitruvian Net

Definition 8.32 (Vitruvian Net) *A* Vitruvian Net *(VN) extends Basic Vitruvian Nets with the notion of scenes and is a 13-tuple* $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, Sc, S, sc_0)$ *where*

1. $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R)$ *is a BVN,*
2. Σ *contains* Sc*,*
3. Sc *is a nonempty set of scene symbols,* $Sc \in \Sigma$*, and* Sc *is disjoint to all other elements of* Σ*,*
4. $S : P \to Sc$ *assigns to each place a scene,*
5. $sc_0 \in Sc$ *is a distinguished element of* Sc*.*

The set of all Vitruvian Nets is denoted by $\mathcal{V}$*.*

$\mathcal{V}$

Remarks:

The type of places $p \in P$, taking values $s \in Sc$, is Sc.

The firing behavior of Vitruvian Nets is unchanged from BVN, because the introduction of scenes does not alter the behavior. All definitions about situations, enabling and the firing behavior remain valid.

Vitruvian System

Definition 8.33 (Vitruvian System) *A* Vitruvian System *is a 15-tuple* $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, sc, S, sc_0, F, M_0)$ *where*

1. $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, sc, S, sc_0)$ *is a Vitruvian Net,*
2. F *is the firing function and*
3. M_0 *is the initial marking.*

With Vitruvian Nets and Systems we have all ingredients to model Vitruv_L specifications. However, we prefer nets where the number of places assigned to scene sc_0 is minimal, because these places do not belong to any scene specified in Vitruv_L and thus any markings of these places model situations which cannot encoded in Vitruv_L explicitly. Also, it is possible that in a Vitruvian System places from different scenes are marked simultaneously. This clearly contradicts our assumption from Vitruv_L that at most one scene is active, hence only places of this particular scene shall be marked simultaneously. Therefore, we constrain Vitruvian Nets and Systems further, to obtain structures and dynamic behaviors respecting the aforementioned requirements.

In a Vitruvian Net places $p \in P$ with scenes $S(p) = sc_0$ are important, because they model the state of the presentation outside a regular scene, e.g. the presentation state at the very beginning and the very end. Additionally, these places can be used for modeling the connection from one scene to another, therefore we call them connector places.

connector place

Definition 8.34 (Connector Place) *Let* $V = (N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, sc, S, sc_0)$ *be a Vitruvian Net. Place* $p_0 \in P$ *is called* connector place *iff*

$$C(p_0) = Sc \quad \wedge \quad S(p_0) = sc_0. \tag{8.36}$$

Modeling a Vitruv_L specification as a Vitruvian Net should result in a net which is as close as possible to original specification. Hence, a minimal set of connector places is of interest, because then the state space outside of regular scenes is then also minimal. We call such minimal Vitruvian Nets regular Vitruvian Nets.

regular

Definition 8.35 (Regular Vitruvian Net) *Let* $V = (N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, Sc, S, sc_0)$ *be a Vitruvian Net. Let* $P_0 = \{p \mid p \in P \wedge C(p) = Sc \wedge S(p) = sc_0\}$ *be the set of connector places. V is called* regular *iff*

1. $|P_0| = 1$*, i.e. we have only one connector place,*
2. $\forall p \in (P \setminus P_0) : C(p) \neq Sc$*, i.e. only the connector place has type Sc.*
3. $Sc = \{S(p) \mid p \in nb(x)\}$ *for all* $x \in nb(p_0)$ *and* $p_0 \in P_0$*, i.e. the set of all scenes is exactly determined by the scenes in the neighborhood of the connector places.*

The smallest regular Vitruvian Net, i.e. where all components are minimal, can be the initial net for an iterative net construction. The following observation deals with that.

Observation 8.36 The smallest regular Vitruvian Net V_0 consists only of the connector place: $V_0 = ((\emptyset, \{p_0\}, \emptyset), (\mathcal{T}), \{\{\bigstar\}, \mathbb{N}, [0,1], D, \{sc_0\}\}, \{p_0 \mapsto Sc\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \{sc_0\}, \{p_0 \mapsto sc_0\}, sc_0\})$.

Proof. We need at least the connector place p_0 to satisfy condition 1 of Def. 8.35. To define p_0 we also need $sc_0 \in Sc$, $Sc \in \Sigma$, and the type $C(p_0)$ and scene $S(p_0)$ of p_0. □

A regular Vitruvian Net is extended to a regular Vitruvian system to model the dynamic behavior closely to the corresponding Vitruv_L specification, where the regular Vitruvian system is a constrained Vitruvian system. First, the regular Vitruvian Nets provides a minimal set of places outside regular scenes. Second, for every marking M in such a regular Vitruvian system, each place p with $|M(p)| > 0$ shall belong to the same scene, i.e. only places of one scene are marked at any time. Additionally, we require the initial marking, M_0, placing only one token into the connector place with a value different from sc_0. This gives some regularity for marking sequence: we start always before any regular scene and the first scene is encoded in the token value of the connector place.

Regular Vitruvian System

Definition 8.37 (Regular Vitruvian System) *A* Regular Vitruvian System *is a 15-tuple* $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, Sc, S, sc_0, F, M_0)$ *where*

1. $(N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, Sc, S, sc_0)$ *is a regular Vitruvian Net,*
2. *F is the firing function and*
3. $M_0 = \{(p, sc)\}$ *with* $p \in P$ *and* $sc \in Sc \setminus \{sc_0\}$ *is the initial marking.*
4. $\forall M \in [M_0\rangle \exists s \in Sc \forall p \in P : |M(p)| > 0 \Rightarrow S(p) = s$, *i.e. all places in each reachable marking from* M_0 *are in the same scene.*

Remarks:

Apparently, the smallest regular Vitruvian Net does not qualify to become a regular Vitruvian system. This is so because $Sc = \{sc_0\}$ allows no marking (p, sc) with $sc \in Sc \setminus \{sc_0\}$, otherwise sc would be an element of the empty set.

8.5.2. Scenes and Leaving Them

The remaining elements of Vitruv_L we have to deal with are scenes and the leave for statement to switch between scenes. The foundation for scenes are prepared in the definition of regular Vitruvian Nets, but the details are still missing. We discuss next the connection of two scenes, as shown in fig. 8.9 on the following page.

As discussed previously, scenes can be considered as a large interval with a definitive start and end point, if we neglect jumps between scenes. All scenes of a presentation have equal rights and are independent of each other. Accordingly, we can simplify scenes and fold them to a transition with exactly one place in its pre- and postset. Each scene can be the very first or the very last scene of the entire presentation, hence we need arcs between the connection place and the start- and end-transition of the scene. A token in the connection place is used for deciding which scene is started.

In fig. 8.9 on the next page we apply these considerations. We have two scenes s1 and s2, denoted as transitions with dashed border, and each one is connected with

the place ConnectionPlace via transitions startS1 and startS2, respectively. These two arcs have arc expressions s1 and s2, i.e. their enabling situations are

$$(\mathsf{startS1}, \{(\mathsf{ConnectionPlace}, \mathsf{s1})\}_{ms}, \{(\mathsf{p1}, \bigstar)\}_{ms})$$
$$(\mathsf{startS2}, \{(\mathsf{ConnectionPlace}, \mathsf{s2})\}_{ms}, \{(\mathsf{p3}, \bigstar)\}_{ms})$$

Therefore, the token in place ConnectionPlace decides which of the scenes is started. After the end of each scene, either transition endS1 or endS2 places a token in ConnectionPlace with value s0, denoting that we have reached a state outside of any scene. If this token is the only marking in the entire net, the net is dead, because no transition is enabled and will ever be enabled. After the end of a scene, nothing else can happen, because we have no means to express the order of scenes in $\mathsf{Vitruv_L}$. The only possibility to activate another scene is to use the leave for statement.

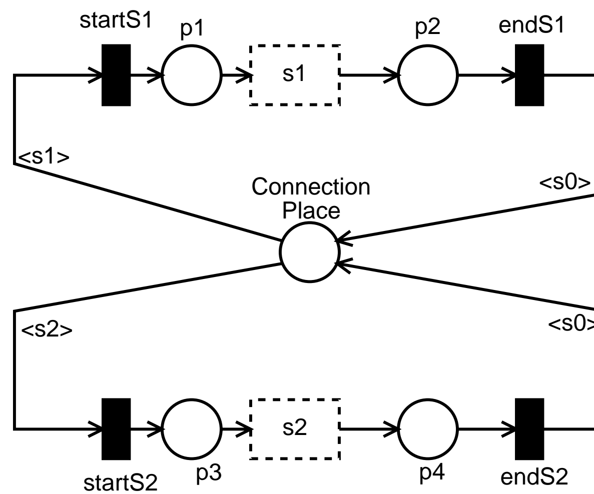

Figure 8.9.: Two scenes with the connection place.

Activation of a leave for statement results in an immediate termination of the current scene and all of its activities, and the target scene is started. With our preparations so far, it is straightforward to define the realization of leave for with regular Vitruvian Nets. Stopping all current activities means removing all tokens from all places, independent of whether or not the tokens have timestamps. Hence, each leave for is realized by a transition with reset arcs to all places in the scene. Its post-set is the place ConnectionPlace, where the transition puts a token with value s_i if s_i is the target scene.

More formally, let $t \in T$ be a leave for transition in scene s_j with target scene s_i. We have reset arcs from all places assigned to s_j to t:

$$\{(p, t) \mid p \in P \wedge S(p) = s_j\} \subseteq R$$

For any marking w_i of $\bullet t$ which enables t, we have the same result of firing function F viz a token in ConnectionPlace with value s_i:

$$F(t)(w_i) = \{(\text{ConnectionPlace}, s_i)\}$$

The sheer amount of reset arcs for leave for transitions makes it hard to draw them explicitly in the graph. Since the post-set of these transitions is also always the same, the resulting graph would be additionally cluttered. Therefore, we introduce as symbol for leave for transitions a new graphical element, a triangle with the target scene as inscription. With this notation, we have a clear interface between scenes, thus it is now possible to split the entire net for a Vitruv_L specification into several subnets, one for each scene.

It should be remarked that this approach is not as general as the introduction of hierarchies in colored Petri Nets (Jensen, 1997), but nevertheless reduces the complexity of a Vitruvian Net, because it is often sufficient to deal with each scene independently.

8.6. Composition of Scenes: Translating Vitruv_L

After presenting the various parts of Vitruvian Nets, we can now discuss how to translate entire Vitruv_L specifications to Vitruvian Nets. The process is driven by the hierarchical structure of scenes dividing scenes into blocks connected by events, selectors and loops. We combine the subnets developed for the various bits and pieces of Vitruv_L into a larger regular Vitruvian Net. The combination works on both, basic and regular Vitruvian Nets.

The combination process follows the hierarchical structure of the Vitruv_L specification, similar to linearization done for generating a Vitruv_I specification from its Vitruv_L counterpart. In contrast to Vitruv_I, where we partition the Vitruv_L specification into a set of independent blocks, we now have to combine these blocks to produce a Vitruvian Net for the entire Vitruv_L specification. During the linearization process, events, loops and selectors are represented as intervals, and loop and selectors bodies as blocks, respectively. We have to keep track which interval and which block represents which event, loop or selector, because we need to identify the proper combinations of subnets to generate the entire net.

With the proper combinations, i.e. with the net representation of the interval, denoting the event, loop or selector, and the subnet representing the interval's internal structure, we replace the interval by the subnet. The replacement is called *expansion*. A recursively applied expansion of events, loops and selectors found in the block of the selector or loop under consideration is called *deep expansion*.

In the following we define how to expand events, loops and selectors. Before that we define some auxiliary functions and how to connect the very first block of scene to a regular Vitruvian net.

8.6.1. Auxiliary Functions

We need some auxiliary functions taking care of tasks required in various expansion and connection processes.

The first auxiliary functions adds a BVN B to a (regular) Vitruvian Net V and a specific scene s. We take the union of the components of both nets and add the places of B to scene function S_V of V. The resulting net is not connected, because it depends on what B models: we have to do different things for events, selectors and loops. We take care of that later on in the specific expansion functions.

add

Definition 8.38 *Let $V = (N_V, (X_j)_V, \Sigma_V, \dots, sc_0) \in \mathcal{V}$ be a Vitruvian Net, let $B = (N_B, (X_k)_B, \Sigma_B, \dots, R_B) \in \mathcal{B}$ be a basic Vitruvian Net, and let $s \in Sc_V$ be a scene of V. The combination of both nets and scene s is defined by function add:*

$$add(V, B, s) = (N_V \cup N_B, (X_j)_V \cup (X_k)_B, \Sigma_V \cup \Sigma_B, \dots, Sc_V, S_V \cup \{(p, s) \mid p \in P_B\}, sc_0\}) \quad (8.37)$$

Remarks:

1. Each corresponding component existing in both nets are joined, the set of scenes, Sc_V, and the distinguished scene, sc_0, remain unchanged. Scene function S_V is extended by the mapping of each place in B to scene s.
2. The union of the universe families requires the index sets J and K to be disjoint: $J \cap K = \emptyset$.
3. Function *add* assumes that no new scene is added and thus the set Sc is not modified by adding the BVN.
4. We take the union of firing functions F_V and F_B, because both nets have no connection. Hence, no modifications of the firing functions are needed.

The replacement of an interval by its respective subnet requires that transitions, places and arcs are removed and new arcs are introduced. Function *modify* takes a Vitruvian Net and modifies the net structure.

modify

Definition 8.39 *Function modify updates the Vitruvian Net $V = (N, (X_j), \Sigma, \dots, sc_0) \in \mathcal{V}$ by removing some places, transitions and arcs, and by adding new arcs.*

$$modify(V, drop_P, drop_T, drop_A, add_A) = V' \quad (8.38)$$

where

$$V' = ((T', P', A'), \dots, Sc'_0) \quad (8.39)$$

$$T' = T \setminus drop_T \quad (8.40)$$

$$P' = P \setminus drop_P \quad (8.41)$$

$$A' = (A \cup add_A) \setminus drop_A \tag{8.42}$$

Elements of V' depending on the modified T', P' and A', such as f', g' etc., are modified accordingly, the other elements are unmodified copies from V.

Remarks:

1. New places and transitions are not needed for function *modify*, because they are introduced via subnets and are added by function *add*.
2. A proper use of *modify* requires that all new arcs only connect transitions and places which are not deleted, i.e. $add_A \subseteq (T' \times P') \cup (P' \times T')$.

If we have to update only one component of a n-tuplet (i.e. of some Vitruvian Net), we adopt the functional override notation introduced earlier for the formal semantics of Vitruv$_l$ (cf. (7.8) on page 137).

Notation 8.40 We notate an update of a single component of Vitruvian Net $V = (N, \ldots, sc_0)$, e.g. Sc, leaving the remainder of V unmodified, with $\oplus$, if it is clear from which component is updated:

$$V' = V \oplus (Sc \cup \{s\}) \tag{8.43}$$
$$= (N, (X_j), \Sigma, C, delay, Sym, f, g, I, R, (Sc \cup \{s\}), S, sc_0) \tag{8.44}$$

Remarks:

1. The notation is also applicable to all other net types.
2. It is possible that the application of $\oplus$ results in an inconsistent net. Consider a regular Vitruvian Net, adding an element s to Sc without using s in the range of function S at the same time violates the definition of a regular Vitruvian Net. However, this is not considered harmful, because the application of $\oplus$ is targeted to be used for more complex net modifications, the task of which is to ensure that all required constraints are satisfied.

Sometimes, and in particular for events, we need access tokens in a place, p, without knowing how many "clients" are needed. Thus, in the initial construction we cannot provide enough copies of p in advance. Therefore we define the function *addClient* adding another "client", p', to place p, returning the modified net. If we have only an empty post-set of p, then *addClient* adds a new transition t_1 as post-set of p as the connection between p and p', i.e. we add the respective arcs between p, t_1 and p' to the set of arcs.

Definition 8.41 *Let $V = (N, \Sigma, C)$ be an abstract Vitruvian Net, p some place in V. Function addClient adds a new place p' to V, such that each token in p is also copied*

addClient

to p'.

$$addClient(V, p, p') = V' \tag{8.45}$$

where

$$\text{if } p\bullet = \emptyset : V' = (P, T \cup \{t_1\}, A \cup \{(p, t_1), (t_1, p')\}, \Sigma, C) \tag{8.46}$$
$$\text{where } t_1 \text{ is a new fresh transition, } t_1 \notin T$$
$$\text{if } p\bullet = \{t_1\} : V' = (P, T, A \cup \{(t_1, p')\}, \Sigma, C) \tag{8.47}$$

with conditions

$$p, p' \in P \tag{8.48}$$
$$C(p) = C(p') \tag{8.49}$$
$$\bullet p' = \emptyset \tag{8.50}$$
$$(p\bullet = \emptyset) \Rightarrow t_1 \notin T \tag{8.51}$$
$$|p\bullet| \leq 1 \tag{8.52}$$

The firing function F for transition t_1 is defined as

$$F(t_1)\{(p, v)\}_{ms} = \{(p_1, v) \mid p_1 \in t_1\bullet\}_{ms} \tag{8.53}$$

Remarks:

1. Since we only require that V is an abstract Vitruvian Net, all other Vitruvian Net types are also allowed.
2. If p has no post-set, we introduce a new fresh transition t_1. If $p\bullet$ is neither the empty set nor the singleton set $\{t_1\}$, it is illegal to apply *addClient*.
3. The place p' needs to have the same type as the original place p.
4. The definition of F for t_1 is the identity function and is independent of the amount of places in the post-set of t_1.

8.6.2. Connecting Scenes

A new scene is connected to a regular Vitruvian Net following the pattern given in sec. 8.5.2 on page 181. In contrast to the example shown there (fig. 8.9), we consider only one scene, s, as BVN B with places p_1 and p_2, being the root and leaf places of B, respectively. We need to introduce transitions t_1 and t_2, which become $\bullet p_1$ and $p_2\bullet$, resp., and connect t_1 and t_2 with the connection place, p_0, of the regular Vitruvian Net, V. In fig. 8.9 on page 182, transitions t_1 and t_2 are called startS1 and endS1, respectively. Firing function F of V needs to be updated also, because we introduce new transitions and new arcs. All these considerations are summarized in the definition of function *connectScene*.

Definition 8.42 (Connecting a Scene) *Let $V = (N_V, \Sigma_V, \ldots, sc_{0,V})$ be a regular Vitruvian Net, $B = (N_B, \Sigma_B, \ldots, R_B)$ be a basic Vitruvian net representing a scene s. Let $p_1, p_2 \in P_B$ be the (unique) root and leaf places of B, and $p_0 \in P_V$ be the unique connection place of V. The connection between scene s represented by B with V is defined by function connectScene introducing new transitions t_1 and t_2, which connect p_0 with p_1 and p_2:*

connectScene

$$connectScene(V, B, p_0, s) = V' \tag{8.54}$$

where

$$\begin{aligned} V' &= ((P_{V''}, T_{V''} \cup \{t_1, t_2\}, A_{V''} \cup \{(p_0, t_1), (t_1, p_1), (p_2, t_2), (t_2, p_0)\}), \\ &\quad (X_j)_{V''}, \Sigma_{V''}, \ldots sc_{0,V''}) & (8.55) \\ \{p_1\} &= root(N_B) & (8.56) \\ \{p_2\} &= leaf(N_B) & (8.57) \\ V'' &= add(V \oplus (Sc_V \cup \{s\}), B, s) & (8.58) \end{aligned}$$

with conditions

$$\begin{aligned} p_0 &\in P_V & (8.59) \\ C(p_0) &= Sc_V & (8.60) \\ C(p_1) &= C(p_2) = \{\star\} & (8.61) \\ s &\neq Sc_V & (8.62) \\ t_1, t_2 &\notin T_{V''} & (8.63) \end{aligned}$$

The enabling situations s_1 and s_2 for t_1 and t_2, resp., – and thereby the firing function – are defined as:

$$\begin{aligned} s_1 &= (t_1, \{p_0, s)\}_{ms}, \{p_1, \star\}_{ms}) & (8.64) \\ s_2 &= (t_2, \{p_2, \star)\}_{ms}, \{p_0, sc_{0,V}\}_{ms}) & (8.65) \end{aligned}$$

Remarks:

1. Connecting a new scene s to an existent regular Vitruvian Net V is done in two steps: first, we add the net B representing s to V after adding s to the set of scenes of V. Second, we introduce new transitions t_1 and t_2 and connect them with p_0, p_1 and p_2.
2. We take simply the union of the firing functions for B and V, because both nets have no connection. For the new transitions t_1 and t_2 we have only one enabling situation for each transition, hence the two situations s_1 and s_2 also determine their firing function.

8.6.3. Expanding Events

When expanding an event we replace the transition denoting the enabling interval of the event and its pre- and post-set by a event subnet, structured as presented in sec. 8.4.2 on page 172. The replacement is carried out by function *expandEvent* the parameters of which are the regular Vitruvian Net with the unexpanded event, the event subnet, the transition of the unexpanded event and the two places of the event subnet denoting start and end of the event's enabling interval. We also need the current scene to add the subnet properly to the regular Vitruvian Net.

Definition 8.43 *Let* $V = (N_V, (X_j)_V, \ldots, sc_{0,V})$ *be a regular Vitruvian Net and* $B = (N_B, \Sigma_B, \ldots, R_B)$ *a basic Vitruvian net which is an event subnet. Let* e *be a transition in* V *denoting an unexpanded event. Replacing the unexpanded event* e *by event subnet* B*, where* Start *and* End *are the named places in* B*, is defined with function expandEvent:*

expandEvent

$$expandEvent(V, B, e, \text{Start}, \text{End}, sc) = modify(V', drop_P, drop_T, drop_A, add_A) \quad (8.66)$$

where

$$V' = add(V, B, sc) \quad (8.67)$$
$$\{e_S\} = \bullet e \quad (8.68)$$
$$\{e_E\} = e\bullet \quad (8.69)$$
$$drop_P = \{e_S, e_E\} \quad (8.70)$$
$$drop_T = \{e\} \quad (8.71)$$
$$drop_A = \{(t, e_S) \mid (t, e_S) \in A_{V'}\} \cup \{(e_E, t) \mid (e_E, t) \in A_{V'}\} \quad (8.72)$$
$$add_A = \{(t, \text{Start}) \mid (t, e_S) \in A_{V'}\} \cup \{(\text{End}, t) \mid (e_E, t) \in A_{V'}\} \quad (8.73)$$

with conditions

$$e \in T_V \quad (8.74)$$
$$\{\text{Start}\} = root(N_B) \quad (8.75)$$
$$\text{End} \in leaf(N_B) \quad (8.76)$$
$$C(e_S) = C(\text{Start}) \quad (8.77)$$
$$C(e_E) = C(\text{End}) \quad (8.78)$$

We modify the firing function for dealing with the new arcs by replacing all occurrences of e_S *and* e_E *by* Start *and* End*, respectively.*

Remarks:

1. The expansion works in two steps: first, we add B to V. Second, we modify the net, such that e and its pre- and post-set is removed from V and all arcs to $\bullet e$ and all arcs from $e\bullet$ are replaced by arcs to Start and from End, respectively.

2. Firing function F from V and B can be joined, because V and B are disjoint. Deleting e and its neighborhood, $nb(e)$, from V' requires modification of the firing function as well: the arcs to and from $nb(e)$ are replaced by arcs to Start and from End, hence the firing function has to reflect these changes by replacing e_S and e_E by Start and End, resp., in all markings.
3. Access to the leaf places of E, e.g. for selectors and loops, is realized by application of function *addClient*, which decouples multiple reactions to the same event as explained on page 174. But this construction is handled during the expansion of loops and selectors.

8.6.4. Expanding Loops

When expanding a loop, we replace the transition denoting the loop and its neighborhood by a loop subnet, structured as presented in sec. 8.4.4 on page 176. The replacement is carried out by function *expandLoop*, the parameters of which are the regular Vitruvian Net with the unexpanded loop, the loop subnet, the transition of the unexpanded loop, the place finished of the loop subnet, the places of event and loop subnet, connecting the loop to its event, and the current scene.

Definition 8.44 *Let* $V = (N_V, (X_j)_V, \ldots, sc_{0,V})$ *be a regular Vitruvian Net and* $B = (N_B, \Sigma_B, \ldots, R_B)$ *a basic Vitruvian net which is a loop subnet. Let* l *be a transition in* V *denoting an unexpanded loop. Replacing the unexpanded loop* l *by loop subnet* B*, where* finished *is a named place in* B*, is defined with function expandLoop:*

expandLoop

$$expandLoop(V, B, l, \text{finished}, p_{1,V}, p_{2,V}, p_{3,V}, p_{1,B}, p_{2,B}, p_{3,B}, sc) = V'' \tag{8.79}$$

where

$$V'' = modify(V', drop_P, drop_T, drop_A, add_A) \tag{8.80}$$

$$V' = addClient(addClient(addClient(V_1, p_{1,V}, p_{1,B}), p_{2,V}, p_{2,B}), p_{3,V}, p_{3,B}) \tag{8.81}$$

$$V_1 = add(V, B, sc) \tag{8.82}$$

$$\{l_S\} = \bullet l \tag{8.83}$$

$$\{l_f\} = l\bullet \tag{8.84}$$

$$drop_P = \{l_S, l_f\} \tag{8.85}$$

$$drop_T = \{l\} \tag{8.86}$$

$$drop_A = \{(t, l_S) \mid (t, l_S) \in A_{V'}\} \cup \{(l_f, t) \mid (l_f, t) \in A_{V'}\} \tag{8.87}$$

$$add_A = \{(\text{finished}, t) \mid (l_f, t) \in A_{V'}\} \tag{8.88}$$

with conditions

$$l \in T_V \quad (8.89)$$
$$C(l_f) = C(\text{finished}) \quad (8.90)$$
$$\mathit{leaf}(N_B) = \{\text{finished}\} \quad (8.91)$$
$$\mathit{root}(N_B) = \{p_{1,B}, p_{2,B}, p_{3,B}\} \quad (8.92)$$
$$P_V \supset \{p_{1,V}, p_{2,V}, p_{3,V}\} \quad (8.93)$$

We modify the firing function for dealing with the new arcs by replacing all occurrences of l_f by finished.

Remarks:

1. Places $p_{1,B}, p_{2,B}, p_{3,B}$ and $p_{1,V}, p_{2,V}, p_{3,V}$ are named Value, End and enabled in fig. 8.8 on page 177 and fig. 8.6 on page 174. They are the interface between event and loop. The nested applications of *addClient* in (8.81) connect these places pairwise, such that Value, End and enabled from loop-subnet B (i.e. $p_{1,B}, p_{2,B}, p_{3,B}$) become a client of Value, End and enabled in V (i.e. $p_{1,V}, p_{2,V}, p_{3,V}$), respectively.
2. In contrast to the event expansion we do not replace arcs to l_S explicitly, because this is handled in the details of the interaction between event and loop subnets.

8.6.5. Expanding Selectors

When expanding selectors, we replace the transition denoting the selector and its neighborhood by a selector subnet, structured as presented in sec. 8.4.3 on page 174. The replacement is carried out by function *expandSelector*, the parameter of which are the regular Vitruvian Net with the unexpanded selector, the selector subnet, the transition of the unexpanded selector, the place Finish of the selector subnet, the places of event and selector subnet, connecting the selector to its event, and the current scene.

Definition 8.45 *Let $V = (N_V, (X_j)_V, \ldots, sc_{0,V})$ be a regular Vitruvian Net and $B = (N_B, \Sigma_B, \ldots, R_B)$ a basic Vitruvian net which is a selector subnet. Let s be a transition in V denoting an unexpanded selector. Replacing the unexpanded selector s by selector subnet B, where* Finish *is a named place in B, is defined with function expandSelector:*

expandSelector

$$\mathit{expandSelector}(V, B, l, \text{Finish}, p_{1,V}, p_{2,V}, p_{3,V}, p_{4,V}, p_{1,B}, p_{2,B}, p_{3,B}, p_{4,B}, sc) = V'' \quad (8.94)$$

where

$$V'' = \mathit{modify}(V', \mathit{drop}_P, \mathit{drop}_T, \mathit{drop}_A, \mathit{add}_A) \quad (8.95)$$

$$
\begin{aligned}
V' &= addClient(addClient(addClient(addClient(V_1, p_{1,V}, p_{1,B}),\\
&\qquad\qquad\qquad\qquad\qquad p_{2,V}, p_{2,B}),\\
&\qquad\qquad\qquad p_{3,V}, p_{3,B}),\\
&\qquad\quad p_{4,V}, p_{4,B}) && (8.96)\\
V_1 &= add(V, B, sc) && (8.97)\\
\{s_S\} &= \bullet s && (8.98)\\
\{s_f\} &= s\bullet && (8.99)\\
drop_P &= \{s_S, s_f\} && (8.100)\\
drop_T &= \{s\} && (8.101)\\
drop_A &= \{(t, s_S) \mid (t, s_S) \in A_{V'}\} \cup \{(s_f, t) \mid (s_f, t) \in A_{V'}\} && (8.102)\\
add_A &= \{(\text{Finish}, t) \mid (s_f, t) \in A_{V'}\} && (8.103)
\end{aligned}
$$

with conditions

$$
\begin{aligned}
s &\in T_V && (8.104)\\
\text{Finish} &\in leaf(N_B) && (8.105)\\
C(l_f) &= C(\text{Finish}) && (8.106)\\
root(N_B) &= \{p_{1,B}, p_{2,B}, p_{3,B}, p_{4,B}\} && (8.107)\\
P_V &\supset \{p_{1,V}, p_{2,V}, p_{3,V}, p_{4,V}\} && (8.108)
\end{aligned}
$$

We modify the firing function for dealing with the new arcs by replacing all occurrences of l_f by Finish*.*

Remarks:

1. Places $p_{1,B}, p_{2,B}, p_{3,B}, p_{4,B}$ and $p_{1,V}, p_{2,V}, p_{3,V}, p_{4,V}$ are named Value, End, enabled and TimeOut, resp., in fig. 8.7 on page 175 and fig. 8.6 on page 174. They are the interface between event and selector. The nested applications of *addClient* connect these places pairwise, similar to (8.81) for the interface between events and loops.
2. In contrast to the event expansion we do not replace arcs to s_S explicitly, because this is handled in the details of the interaction between event and selector subnets.

8.6.6. Net Construction

With these preliminaries we can now outline an algorithm to derive a Vitruvian Net V from a Vitruv_L-specification L. This is shown in fig. 8.10 on page 193. The basic idea of the algorithm is for each scene to walk through the hierarchy of blocks in

the current scene and to combine blocks by adding subnets for events, selectors and loops. Finally, all leave for transitions are connected with place connectionPlace.

The algorithm assumes implicitly to have access to a series of mappings which can be inferred from the translation process of Vitruv_L. Essentially these mappings are parts of the augmented abstract syntax tree and the accompanying symbol table. In particular, we need to keep track the nesting of blocks as determined in Vitruv_I, the connection of blocks with events, loops and selectors as determined in Vitruv_L. We also need to maintain a list, U, of unexpanded events, loops and selectors, because this list controls the next expansion steps.

Let us now consider the steps in more detail. We start with the minimal regular Vitruvian Net consisting only of the connection place, p_0, which is our entry point. After that we initialize each scene: we model each scene as a Vitruvian Net, which we connect with p_0. This is carried out in function *connectScene*. The Vitruvian Net representing the scene is unexpanded, i.e. all events, loops and selectors are only modeled as simple intervals as we have also done in Vitruv_I. We collect these unexpanded events, loops and selectors in list U. The next steps carry out the deep expansion of the scene. We do a breadth-first traversal through the block hierarchy of the scene. For each element of list U, we generate an appropriate unexpanded subnet and replace the simple unexpanded interval by the corresponding subnet in steps b), c) and d). The replacement is done by functions *expandEvent*, *expandLoop* and *expandSelector*. After each expansion, the corresponding unexpanded element is deleted from U. Since the expansion works only on one level at each step, each expansion might introduce new unexpanded events, loops and selectors, which are added to U'. If after step d) U' is not empty, we move all elements of U' to U in e) and start with the next cycle of the expansion loop working on the elements in U, thereby recursing deeply into the scene definition.

The process terminates since each block hierarchy of a scene is finite and each expansion only adds new child-blocks, the internal structure of which is inaccessible for their parent-blocks. Hence, the expansion of a child block cannot modify the parent-block, which guarantees that the expansion process terminates in the leaf-blocks. After all blocks of a scene are expanded we do the same for the next scene. Finally, after all scenes are connected with p_0 and are expanded, we model the leave for statements. Each transition, t, representing a leave for is connected with p_0, i.e. $t\bullet = \{p_0\}$, and a reset arc from each place in the current scene to transition t is established. After that the net is complete.

In sec. 10.4.2.3 on page 246 we show an application of this algorithm in a larger example.

1. Start with the minimal regular Vitruvian Net, V, consisting only of place ConnectionPlace, here called p_0.
2. for each scene *sc* do:
   a) translate the block of *sc* into BVN B, and connect B with V:
   $V \leftarrow \textit{connectScene}(V, B, p_0, sc)$
   and establish the list U of unexpanded events, loops and selectors of *sc*, list U' is empty.
   b) for each unexpanded event e in U do:
      i. generate the event subnet E for e (see sec. 8.4.2 on page 172) with its distinguished places Start and End,
      ii. extend V with E, after that e is expanded and deleted from U:

      $$V \leftarrow \textit{expandEvent}(V, E, e, \text{Start}, \text{End}, sc)$$

   c) for each unexpanded loop l in U do:
      i. generate the loop subnet L for l with the distinguished places finished, Value_L, End_L and enabled_L (see fig. 8.8 on page 177, there without subscript). L includes BVN B for the (unexpanded) loop body. Identify in V the distinguished places Value_V, End_V and enabled_V from the event subnet for l.
      ii. extend V with L, after that l is expanded and deleted from U:

      $$V \leftarrow \textit{expandLoop}(V, L, l, \text{finished}, \text{Value}_V, \text{End}_V, \text{enabled}_V \\ \text{Value}_L, \text{End}_L, \text{enabled}_L, sc),$$

      iii. extend list U' of unexpanded events, loops and selectors by the respective elements of B.
   d) for each unexpanded selector s in U do:
      i. generate the selector subset S for s including BVNs B_i for the (unexpanded) bodies of the alternatives paths. Identify in S the distinguished places finished, Value_S, End_S, enabled_S and TimeOut_S (see fig. 8.7 on page 175), and in V their counterparts from the event subnet Value_V, End_V, enabled_V and TimeOut_V.
      ii. extend V with S, after that s is expanded and deleted from U:

      $$V \leftarrow \textit{expandSelector}(V, S, s, \text{Finish}, \text{Value}_V, \text{End}_V, \text{enabled}_V, \text{TimeOut}_V \\ \text{Value}_S, \text{End}_S, \text{enabled}_S, \text{TimeOut}_S sc)$$

      iii. extend list U' of unexpanded events, loops and selectors by the respective elements of B_i.
   e) if there are any unexpanded events, loops or selectors in V, i.e. U' is not empty, then let U be U', clear U' and goto b).
3. for each leave for transition add the arc to p_0 and reset arcs to all places from their scene (see sec. 8.5.2 on page 181).

Figure 8.10.: The Algorithm for Connecting Nets

9. Specifying with Natural Language

In the previous chapters we discussed the formal models of the Vitruv approach. We now outline how it is possible to use specification techniques based on natural language within this approach.

9.1. Basic Considerations

After discussing Vitruv_L and its formal semantics in the previous chapters, we are now able to present Vitruv_N, the natural language (NL) based specification language.

The purpose of Vitruv_N is to allow non-technical developers to understand specifications of multimedia presentations with ease, in particular without technical training. In addition, the use of NL makes it much easier for them to write specifications as well. Since the semantics of Vitruv_N are defined via Vitruv_L, we have the same central features in Vitruv_N as in Vitruv_L, viz:

- temporal relationships including fuzzy quantitative annotations,
- scenes and media objects for modularization,
- interactivity with events.

In our opinion defining classes and algorithms should better be left to technical developers. Therefore, classes appear in Vitruv_N only implicitly (see the mapping to Vitruv_L in sec. 9.3 on page 207), and – compared with Vitruv_L – interactivity is available in a simplified form only (see sec. 9.2.4.2 on page 205). For describing multimedia presentations, however, these simplifications should mostly be sufficient.

As discussed earlier, NL as basis for a specification language requires to deal properly with the inherent problems of NL, i.e. with ambiguity, imprecision, vagueness and incompleteness. Imprecision and vagueness are handled within the semantics of Vitruv_N, i.e. in Vitruv_L and its formal semantics, by using fuzzy set theory for denoting vague or imprecision statements. Also incompleteness of Vitruv_N specifications is tackled in the formal setting, since the translation of the Vitruv_N specification to Vitruv_L indicates missing elements and with Vitruv_L's formal semantics we can reveal whether the specification is lacking required information.

The problem of ambiguity cannot be handled via the semantics, but is linked to language structure. Essentially, it is connected with the central design decision for

specification language based on natural language: either we allow the free use of NL or we restrict ourselves to a limited subset of NL. The former alternative is, in general, more user friendly, since it does not prevent users applying any kind of NL they like. There are no restrictions on vocabulary or grammatical structures. Concerning tool support (see req. 4 on page 11), this approach relies on parsing techniques from language understanding, which have their difficulties. The aforementioned approaches from Rolland and Proix (1992) or Gervasi and Nuseibeh (2002) have applied such techniques with some success, but they either generate only a linguistic model of the specification or are not concerned with the complete comprehension of the specification. If we use a limited subset of NL only, we have a different set of pros and cons. The subset of NL can be defined by a restricted vocabulary or a restricted grammar (or both). Apparently, it constricts the specifier's freedom concerning the language used. For writers it might be an error-prone process to write correctly. Using sophisticated editing tools can ameliorate the latter drawback by preventing many errors in advance. On the positive side, a restricted grammar dramatically eases construction of analyzing tools, since we can move from general rewriting systems towards more standard compiler techniques. Concerning ambiguity, the restricted grammar and vocabulary makes it more easy to allow only such sentence structures which avoid ambiguous situations. This is an additional reason why we prefer the restricted option.

Another crucial point in designing a specification language based on NL is how to deal with the semantics of the specification language. In particular, the formal semantics should preferably not be visible in the specification language, since the main motivation to use NL is to support non-technical developers, which are not interested in formal semantics at all. However, this approach is also applied in formal language design. For instance, most functional programming languages, although based on lambda calculus, do not support lambda calculus in its pure form but apply so-called "syntactic sugar" (e.g. Mitchel, 1996; Schmidt, 1994; Winskel, 1993), choosing semantical equivalent but better human-readable formulations. In contrast to formal language design, where it is sufficient to define the semantics for each language construct explicitly, in our case we have to ensure that the intuitive semantics match the formal semantics as close as possible. Only in this case, non-technical developers can understand documents written in the specification language with everyday reasoning and without (formal) training.

For Vitruv_N, we assume that specifications are more often read than written, and thereby it is important to support an easy and comfortable reading. But we also want to make tool development possible, and thus opt for a context-free grammar of a careful selected subset of NL, which is amenable to relative efficient parsing. To satisfy both expectations, we have to provide a language style which is not too tedious to read by offering alternative formulations of the same semantic issues, i.e. we provide different pieces of syntactic sugar for the same semantic element.

After these basic considerations, we discuss the selected language elements in more

detail in the next section.

9.2. Language Elements

To give Vitruv_N a proper background regarding multimedia presentations, we choose storyboards as the main metaphor for the language design. Storyboards originate from the movie industry and are a well-known means for sketching multimedia presentations (Bailey et al., 2001a; Harada et al., 1996). Often a basic screen layout is sketched for each scene combined with annotations of the action. The storyboard metaphor structures specifications in Vitruv_N, since each scene needs to be described and each scene description has a similar structure, listing its constituents and their temporal behavior. Therefore, the storyboard metaphor establishes important stylistic and structural features of the language used.

Vitruv_N is based largely on concepts and structures found in the German specification language SFMP (Specification language For Multimedia Presentations), proposed in the diploma thesis of Matthias Heiduck (1999). In an earlier paper (Alfert and Heiduck, 2002), we presented a first proposal for translating SFMP into English and for giving SFMP a more formal basis. Vitruv_N, as presented in this thesis, is a newer development of this proposal. For SFMP, Heiduck (1999) shows a context-free grammar definition, therefore a similar grammar should be definable for Vitruv_N as well. Of course, it is not guaranteed that parsing the grammar can be very efficiently done, it may be required that we use Earley's algorithm.

Although based on the same roots, there are some differences between SFMP and Vitruv_N. Most obviously, SFMP is a German specification language, whereas Vitruv_N is in English. Additionally, we restrict ourselves in Vitruv_N to the temporal behavior, as found in Vitruv_L, and thus omit spatial specifications for layout descriptions and the media production hints found in SFMP. These differences are of minor concern for this thesis, since we are only interested in specifications of temporal behavior and their support for non-technical developers. The temporal features are maintained in both languages, SFMP and Vitruv_N.

In the following, we present the elements of Vitruv_N with the help of some examples.

9.2.1. Preliminaries

The following description of Vitruv_N documents assumes a simple text-only format of the specification documents. Usually, such documents have a rich format, e.g. for emphasizing headings etc. Under usability considerations such formatted documents are necessary, however, for our purposes they are not required. As a consequence, any embellishments found in Vitruv_N specifications have only commenting characteristics

but do not have any formal semantics, or with other words: the embellishments are intended for the human reader only.

Similar to ordinary programming and specification languages, Vitruv$_N$ requires to identify entities by name. Such names (or identifiers) may be several words long. Identifiers in Vitruv$_N$ are delimited by quotes and may remind the reader of strings in programming languages. This rather unusual form, at least from a programming language viewpoint, has the benefit that identifiers may use otherwise reserved words without conflict, since it is always perfectly clear whether we have an identifier or another part of the specification. This is in contrast to languages such as PL/1, where the decision whether a token is a reserved work or an identifier is context dependent (Waite and Goos, 1984, p. 136), e.g. in the following code snippet: IF IF THEN THEN ELSE ELSE.

For quantitative duration measures we can apply the usual time units of hours, minutes, seconds and milliseconds. Fragments of hours and minutes are as usual given as 1/60 of hours and minutes and are separated by colons, whereas fragments of seconds are given as decimal fractions and separated by a decimal point. A duration of 23 minutes, 37 seconds and 415 milliseconds is denoted as 23:37.415 minutes. For differentiating between precise and fuzzy durations we use as heuristic the use of fragments: a duration of 5 seconds is considered as fuzzy duration whereas 5.0 seconds are a precise duration. Derivations from this heuristic can be resolved in the binding of the specification after a translation from Vitruv$_N$ to Vitruv$_L$.

To distinguish code fragment in Vitruv$_N$ from ordinary text we use angle brackets: ⟨This is an Vitruv$_N$ example⟩, identifiers are shown with their quotes. In the following examples we use three dots ⟨. . . ⟩ as abbreviation for omitted parts of the specification. Text in italics is considered as a comment and is often found as an explanation of the abbreviation. Both, the dots and the comments are not part of the syntax of Vitruv$_N$.

9.2.2. Presentation and Scenes

Each Vitruv$_N$ document specifies one multimedia presentation, consisting of a list of scenes, the first of which starts the presentation. Scenes are the top-level structuring mechanism in Vitruv$_N$, and each scene has its own scope for its constituting elements. Scenes are named with an identifier, which can be used to reference scenes, e.g. for following links.

In example 9.1 on the facing page we show a presentation definition including one scene. The beginning of the presentation is introduced by ⟨Beginning of scene definitions:⟩, followed by a list of scene definitions. The presentation is closed by the single word ⟨End.⟩. Each scene definition starts with ⟨Description of scene⟩, followed by an identifier (here: ⟨“The first scene”⟩). After that, a statement about the duration of the scene may follow. The duration can be given qualitatively (e.g. by stating it is ⟨short⟩ as in the example) or quantitatively. The scene definitions is finished by ⟨End of the

scene description.⟩.

Specification 9.1 Scene Definitions

Beginning of scene definitions:

Description of scene "The first scene".

It is a short scene.

. . . details about the scene

End of the scene description.

. . . additional scene descriptions may follow

End.

The details of a scene consist of a list of media and their composition. This is discussed in the next sections.

9.2.3. Media Definition

In each scene a list of media is used. The media are declared and defined in the first part of a scene definition, assigning to Each media an identifier and a type. The list of types is open, we only require that for each type a respective class in Vitruv_L exists. The five types ⟨video⟩, ⟨audio-clip⟩, ⟨image⟩, ⟨text⟩ and ⟨button⟩ are defined in the prelude of Vitruv_L and thus are the default set of media types.

In spec. 9.2 on the next page we show the media definitions for a scene. It starts with ⟨In this scene, these media are used:⟩, followed by the media definitions. The list of definitions is closed by ⟨End of media definition.⟩. Each media can be characterized by an adjective, given between the article and the media type (here: loud). Additionally, a short *content description* can be given, as shown for the text media. Both, this adjective and the content description are intended for the human specifier only and are ignored in the mapping to Vitruv_L. For defining the identifier of media, different formulation variations can be used as shown in the example. For videos and audio-clips we can specify additionally their duration.

content description

Media with a temporal extension, i.e. videos and audio-clips, may have a (temporal) structure. A typical example is a video with different shots or topics. In the Altenberg Cathedral context it might be a video showing several cathedrals, where we want to identify for each cathedral when it is shown. In Vitruv_N, these parts of media are called *content elements* and can be specified as part of the media definition.

content elements

In spec. 9.3 on the following page we see the definition of two content elements of video ⟨French⟩. The definition of content elements is introduced by ⟨In this media the following content elements exist:⟩ and is finished by ⟨End of the content description⟩.

Specification 9.2 Media definition

In this scene, these media are used:

- The video, identified by “the video”. It has a duration of about 10 seconds.
- A loud audio-clip, in the following named as “loudness”.
- A detailed text about French cathedrals, identified by “cathedrals”.

. . . additional media definitions may follow

End of the media definition.

---

element composition

Each element definition starts with ⟨An element⟩. The temporal arrangement of content elements with respect to each other and the containing media can be defined in the ***element composition***. Here we can use the same rules as for the media composition, explained in the sec. 9.2.4, with the restriction that only references to the media itself and its content elements are allowed. The element composition finishes the content description and is introduced by ⟨Element composition:⟩.

Specification 9.3 Content Elements

The video with filename “French.mov”, in the following named as “French”. In this media the following content elements exist:

An element named “Chartres”.

An element identified by “Amiens”.

. . . more elements may follow

Element composition:

The element“Chartres” starts immediately after the beginning of the video. The element “Amiens” is 30 seconds long and follows “Chartres”.

End of the content description.

9.2.4. Media Composition

The media composition defines the temporal arrangement of the various media used in a scene. It starts with ⟨The media composition:⟩ and ends after ⟨End of the media composition⟩.

9.2.4.1. Temporal Arrangement

Defining the temporal arrangement of media is based on two central concepts, intervals and time points. If we address the entire media, then we use the concept of an interval since the media has temporal extension. If we refer to the start or end of a media, then we are talking about time points. The distinction is important, because the relations between intervals and time points are different from relations between time points and relations between intervals and should be reflected by the language design.

The media composition section is the central part of Vitruv_N and in contrast to other parts (cf. the media definition) it is less structured than a list of entities. Therefore, the requirement of good readability is of particular importance here and thus we provide a variety of sentence structures. In the following enumeration we list them with a high-level grammar and an example. The nonterminals used in this grammar are explained in table 9.1 on the following page, alternatives are separated by | similar to regular expressions. With TE we identify a temporal entity, which is either a media, a content element, or a time point of a media or of a content element. Obviously, duration constraints are not allowed for time points, since their temporal extension is zero.

1. TE RV | RP TE: ⟨The scene starts with video “French”.⟩ The relation is either given as a verb or as a preposition.

2. RP TE TE: ⟨During video element “Amiens” we hear the audio-clip “present Amiens”⟩. Between both TE we have to add an appropriate activity, here ⟨we hear⟩, to form a proper sentence.

3. TE ‘and’ TE RV: ⟨Video “French” and audio-clip “Bells” start together⟩. Both TE are subject of the verb. The word ‘and’ in the grammar is a literal.

4. TE DURATION ‘and’ RV | RP TE: ⟨The audio-clip “Bells Intro” is short and is followed by audio-clip “Chartres Explained”.⟩ The word ‘and’ in the grammar is a literal.

5. TE DURATION: ⟨Button “Abort” is enabled for 25 seconds.⟩

Beside the duration specification we can also qualify each media or content element with a qualitative duration, e.g. ⟨The scene starts with the long video “French”⟩. We present a media composition part of a scene definition applying these various grammatical structures in spec. 9.4 on the next page.

The relations used in the grammar relate pairs of time points, pairs of intervals, or a time point and an interval, respectively. The relations between time points are either ⟨before⟩, ⟨after⟩ or ⟨equals⟩. Between a time point and an interval we have ⟨before⟩, ⟨after⟩ and ⟨during⟩, if we reverse interval and time point then we have instead of

Specification 9.4 Media composition without events

The media composition:

The scene starts with video “French”. The audio-clip “Bells Intro” is short and starts at the beginning of video “French”. The audio-clip “Bells Intro” is followed by audio-clip “Chartres Explained”. During video element “Amiens” we hear the audio-clip “present Amiens”. Button “Chartres Button” is active parallel to the video element “Chartres”.

. . . more media composition descriptions may follow

End of the media composition.

nonterminal	*meaning*
TE	timed entity, i.e. an interval or time point
RV	relation verb, i.e. a temporal relation as verb
RP	relation preposition, i.e. a temporal relation as preposition
DURATION	a duration specification, either quantitative or qualitative

Table 9.1.: Nonterminals of the grammar for temporal specifications.

⟨during⟩ relation ⟨contains⟩. If we relate a time point to the beginning or end of an interval, we do not have a relation between an interval and a time point but rather between two time points, since start and finish of intervals are considered a time point.

For relations between intervals we could use the 13 well known relations defined by Allen (1983). We made experiments with users, including an English native speaker without a computer science education, applying a first version of Vitruv_N on scene ⟨The Intro⟩, presented later in sec. 10.2.1 on page 220. The experiments showed that Allen’s relations do not always satisfy the semantics of everyday English. Therefore, we chose more appropriate terms and formulations, which are sometimes quite contrary to Allen’s approach. They, however, meet the expectations of the users more correctly, which is what is required here. In table 9.2 on the next page we summarize allowed formulations together with their translation into Allen’s calculus.

Interesting is that some of the formulations look like a modified relation or a combination of relations (e.g. ⟨starts immediately after⟩), but they are in fact linguistic variants of completely different relations in Allen’s calculus (here: met by). Another interesting property is shown for ⟨starts with⟩, which translates to started by in Allen’s system of relations. This is because Allen requires that the first mentioned interval of the ⟨starts⟩ relation is the shorter of both intervals. Thus, ⟨starts with⟩ and Allen’s starts are inverse relations, since the ⟨starts with⟩ implies that the second interval is

the shorter of both intervals. Other relations, such as ⟨contains⟩, are in their NL variant usually indifferent whether the borders are included. Therefore, we differentiate between ⟨contains⟩ and ⟨contains completely⟩, similar to the subset and proper subset relations in set theory.

Natural Language Relation	*Relation in Allen's Calculus*
A before B	A before B
A after B	A after B
A meets B	A meets B
A and B start together	A (starts or is started by or equal) B
A and B finish together	A (finishes or is finished by or equal) B
A and B overlap	A (overlaps or is overlapped by) B
A starts before B	A (before or overlaps) B
A starts B	A meets B
A starts immediately after B	A is met by B
A starts with B	A is started by B
A finishes before B	A before B
A finishes earlier than B	A (before or overlaps or starts) B
A finishes immediately before B	A meets B
A finishes B	A is met by B
A during B	A (starts or during or finishes) B
A within B	A (starts or during or finishes) B
A completely within B	A during B
A contains completely B	A contains B
A contains B	A (is started by or contains or is finished by) B
A overlaps B	A is overlapped by B

Table 9.2.: Natural language expressions regarding intervals and their counterparts in Allen's calculus.

All these relations can be qualitatively modified, e.g. the relation ⟨after⟩ can be modified to ⟨shortly after⟩, where the modifier ⟨shortly⟩ add some detail about the distance between both related entities.

In addition the aforementioned temporal relations we have some other relations as well, which are not built from words of Allen's system of relations. We call these new relations *synonyms*, since they can (or better: must) be replaced by some of Allen's relations to define their semantics, similar to synonyms in linguistics. Therefore, synonyms can be seen as macros, which are replaced by their definition when we transform the Vitruv_N specification to Vitruv_L (see sec. 9.3.4.1 on page 212). The synonyms

synonyms

used in the examples are listed in table 9.3, where we also present their definition in terms of Allen's relations.

We call the representation of relations in Vitruv_N in terms of Allen's relations as the relations' *canonical form*.

canonical form

Synonym	*Relation in Allen's Calculus*
A follows B	A met by B
A parallel to B	A equals B
A at the same time as B	A equals B
A goes beyond B	B (overlaps or starts) A
A extends beyond B	B (overlaps or starts) A

Table 9.3.: Some synonyms for temporal relations

Similar to the relations, we can also use modified qualitative duration constraints. This means that we not only allow single adjectives such as ⟨short⟩ or ⟨long⟩ as qualitative durations, but also expression such as ⟨very long⟩ or ⟨not too long⟩. Some of these modified durations can immediately be built from the set of modifiers defined in Vitruv_L (e.g. ⟨very long⟩ or ⟨extremely short⟩), while others (e.g. ⟨not too long⟩) do not have this feature and thus are similar to the synonyms of interval relations: they need an explicit translation definition. Because of this structure we use the same terms of modifiers and synonyms for the additional duration constraints as for the relations. In table 9.4 we show the three modified durations presented in this paragraph and their definitions as Vitruv_L expressions.

Modified Duration or Synonym	*Duration constraint in* Vitruv_L
very long	very (DURATION.long)
extremely short	extremely (DURATION.short)
not too long	below (DURATION.long)

Table 9.4.: Some modified durations and synonyms

However, we do not fix the set of modifiers and the set of synonyms for relations and durations in Vitruv_N for various reasons:

- Modifiers and synonyms are crucial for creating specifications, which are not boring to read because of a limited set of allowed wordings. By not fixing the set, it is easy to support specific dialects of developer groups.

- The intuitive semantics of modifiers and synonyms should be clear from the context for the human reader of the specification. Due to their macro characteristics it is easy to provide a library of definitions as compound relations for them as part of the Vitruv_L prelude, which can used for a (not necessarily automatic) translation from Vitruv_N to Vitruv_L. Any further details are left to an implementation.

9.2.4.2. Interaction

We have now to discuss how to describe interactions in Vitruv_N, which requires to describe events and reactions to events. In contrast to Vitruv_L, more complex behaviors, in particular combinations of loops and selectors, shall not be our primary concern for Vitruv_N. Such behavioral descriptions are better left for technical developers. Therefore, we do not support loops and selectors in full detail, but only in a rather simple form.

In Vitruv_N we have four different types of events: button can be pressed, and for mouses we have click, double click and rollover events. Mouse events can be used with every media or content element, button events only with buttons. The activation of an event is then implicitly equal to the activation of respective media. For simplicity reasons, the reaction to an event is either an ⟨if⟩-statement or a ⟨loop⟩, but not both. This means that we do not support multiple reactions to an event, which however are allowed in Vitruv_L. Any ⟨if⟩-statement refers to exactly one action in its consequent part, which either follows a link to another scene or starts a media. In spec. 9.5 we present an ⟨if⟩-statement which links the button ⟨ChartresButton⟩ to the scene ⟨Chartres Info⟩ which must be defined elsewhere in the same specification. Alternative formulations for the activation of scene ⟨Chartres Info⟩ are ⟨leave for scene ...⟩ (as in Vitruv_L) and ⟨replace the current scene by scene ...⟩.

Specification 9.5 Branching to another scene

... declaration of "ChartresButton"

The media composition:

...

If button "ChartresButton" is pressed then scene "Chartres Info" is activated.

...

End of the media composition.

Loops have a temporal extension and are thus considered an additional media type. Therefore, loops need to declared in the media definition section as of type ⟨Loop⟩ before we can use them in the media composition section. This is similar to loops

in Vitruv_L (see sec. 5.5 on page 70). In spec. 9.6 we present a loop ⟨L⟩ depending on button ⟨b⟩. This specification corresponds to spec. 5.7 on page 73: loop ⟨L⟩ iterates until button ⟨b⟩ is pressed and plays in each iteration the audio-clip ⟨a⟩. After the loop finishes, video ⟨x⟩ is activated. The body of the loops is introduced by ⟨Loop . . . repeats until . . . :⟩ and is closed by ⟨End of the loop⟩. In the body of loops arbitrary media compositions are allowed, but as in Vitruv_L, and for the same reason, the used media must not be applied outside the loop body.

Specification 9.6 Loops in Vitruv_N

A loop named "L".

. . . declaration of b, a, x . . .

The media composition:

The scene starts with activating button "b". Loop "L" repeats until button "b" is pressed:

The loop starts with audio "a".

End of the loop.

Loop "L" is followed by video "x".

End of the media composition.

9.2.5. Omitted Elements of Vitruv_L

Not every part of Vitruv_L specifications is expressible in Vitruv_N, since some of them are either better suited for technical developers or they do not occur in a NL specification. We will list these issues here and discuss them briefly.

- An explicit binding section is not available in Vitruv_N, because we are interested in a conceptual model and not its realization. Some information in Vitruv_N specifications can be used for establishing a binding, when the specifications are is translated to Vitruv_L. In particular, all quantitative duration constraints and each declared and used media object occur also in the respective Vitruv_L binding.
- While instances of classes defined in Vitruv_L are available in Vitruv_N, e.g. for media types, class definitions cannot be made. The required knowledge about class hierarchies, syntactic and semantic details of inheritance, etc., prevents the explicit use of object-orientation without training. Although possible NL statements, like "X is a Y", give a hint for applying inheritance, the intended semantics of such statements, however, are too ambiguous. Typical pitfalls, such as incompatible class hierarchies (Do unbounded stacks inherit from bounded

stacks or the other ways round or neither?) or mixing of classes and instances, can arise easily. Therefore, we prefer limited expressiveness of $Vitruv_N$ in favor of a unambiguous specification.

- Additional compound relations need not to be defined in $Vitruv_N$, but can simply be used. Their definitions, however, have to be provided for $Vitruv_L$, since we need the definitions for the technical realization. But common sense should be enough to understand what their meaning is, hence no definition in $Vitruv_N$ is required.
- New fuzzy types can not be defined in $Vitruv_N$. They are considered as too technical and thus, similar to classes, are omitted from $Vitruv_N$.

9.3. Mapping $Vitruv_N$ to $Vitruv_L$

In this section we explain how to translate systematically a $Vitruv_N$ specification to $Vitruv_L$ in a top-down manner, following the structure of sec. 9.2. While we do not present an algorithm in its entirety, we provide the core features of the translation algorithm which should provide enough information to perform the transformation by hand. We illustrate the process by translating the examples given earlier in sec. 9.2. Additionally, in sec. 10 on page 219 we show a larger example of a $Vitruv_N$ specification with its transformations into $Vitruv_L$ and Vitruvian Nets.

The central idea of the transformation is to preserve as much structure, names, etc. as possible from the source specification into the transformed $Vitruv_L$ specification. We do this by applying a three-level scheme for scenes and media:

1. each scene becomes a class,
2. each media ⟨o⟩ of type ⟨m⟩ defined in a scene becomes an instance of a predefined media class M if media ⟨o⟩ does not contain any content elements. Otherwise, we generate a class M' of its own for media ⟨o⟩and instantiate class M'.
3. each content element is modeled as an instance from class Interval.

These classes are augmented by event declarations if needed. Additionally we construct a rudimentary binding section indicating the media and scenes used.

This should give an impression of the path of the transformation process, we will now investigate the details.

9.3.1. Preliminaries

Some general rules can be discussed independently of the transformation context. In particular the handling of identifiers and duration measures are to mention.

Whitespace is not significant for identifiers in Vitruv_N. This makes it easy to transform these identifiers to Vitruv_L, since we can simply remove any whitespace to get appropriate identifiers for Vitruv_L. For better readability, we can choose whether we replace whitespace by underscores or start each new word with a capital letter: the identifier several words long would replaced to several_words_long or to severalWordsLong, respectively. However, this is a matter of taste. If an identifier name clashes with a reserved word from Vitruv_L or predefined identifier, e.g. from the prelude, a completely fresh identifier should be chosen. In the following discussion we use identifiers from Vitruv_N with their quotes to distinguish them unambiguously from identifers of Vitruv_L specifications.

In Vitruv_L we do not have the notion of time units, there is only the real line as abstraction of time. Therefore, it is required that a transformation from Vitruv_N to Vitruv_L fixes a mapping from the various time units into the real line, i.e. into a system with only one unit. For simplicity reasons, we propose to use seconds as unit of choice and apply this mapping throughout all examples. However, which specific mapping is used does not matter (at least semantically), the mapping needs only to be fixed for each transformation process. Any further details are left to an implementation.

9.3.2. Presentations and Scenes

Each scene defined in Vitruv_L is mapped to a class in Vitruv_L which inherits the marker class Scene (see 5.6 on page 78). All media defined in the scene become elements of the scene class. In the binding, each scene is instantiated once. Essentially, the scene classes and most other classes, created during this transformation, are singleton classes.

The first scene in Vitruv_L is the first scene in the presentation, therefore the instance of the respective class is also the first scene in binding section.

In spec. 9.7 on the next page we see the transformed specification of scene ⟨“The first scene”⟩ (taken from spec. 9.1 on page 199). This specification comes complete with binding, but the more interesting parts are left to later specifications, after we introduce the remaining transformation rules.

9.3.3. Media Definitions

The media defined in a scene are transformed to elements of the respective scene class. The type of these class elements corresponds to the media type of Vitruv_N: media type ⟨video⟩ corresponds to the predefined class Video, and so on. If we have a duration constraint for the media (e.g. ⟨a long video⟩), then this constraint is transformed to a duration constraint on the length of object in the body of the scene class. Additionally we add an entry for each media in the binding making sure that an instance of respective class is created. These entries in the binding may be empty, e.g.

Specification 9.7 The first scene in Vitruv$_L$

```
// additional definition classes, compound relations and fuzzy types
class TheFirstScene extends Scene
  let
    // some media definitions
  body
    this.length is DURATION.short;
    // further relations and constraints
  end;
end;
// additional definition classes, compound relations and fuzzy types
bindings
  in prelude
    // binding of the prelude
  end;
  object tfs : TheFirstScene
    // binding of the first scene
  end;
  // binding of additional scenes
end;
```

object theVideo : Interval end;. This is the case whenever we do not have any information about the internals of the media. Nevertheless, this is sufficient for instantiating the respective objects.

In spec. 9.8 on the next page we see the respective parts of a Vitruv$_L$ specification, based on the spec. 9.2 on page 200. The three media definitions of ⟨"the video"⟩, ⟨"loudness"⟩ and ⟨"cathedrals"⟩ are modeled as elements theVideo, loudness and cathedrals, resp., of an appropriate class. In the binding, we realize the duration constraint (⟨ca. 10 seconds⟩) of ⟨"the video"⟩ as triangular fuzzy set with 10 as core value.

If the media contains media elements then the construction is more complicated. The predefined media classes do not have any internal structure and thus cannot be used to model the media with its elements. Therefore, we create a new class for the media inheriting from the respective class of the media type. All media elements become elements of the new class with type Interval, since we do not know anything specific about the media elements. All qualitative duration constraints and temporal relationships between the elements and the media are transformed into the body of the newly created class. Quantitative duration constraints make their way to the binding of the respective instance of the new class.

The respective media is usually the only instance of its class, since each media is unique, because no other media with the same structure exists. This is the reason why we do not require any kind of equivalence tests of media structure during the trans-

Specification 9.8 Media definitions in Vitruv_L

```
// in some class extending Scene
let
  theVideo : Video;
  loudness : Audio;
  cathedrals : Text;
body
... // the remainder of the specification
bindings
...
  object theVideo : Video
    let length := triangle (6, 10, 14);
  end;
  object loudness : Audio
  ...
  end;
  object cathedrals : Text
  ...
  end;
end;
```

formation, even if their outcome could result in fewer classed generated. Of course, the assumption that no equivalent structures exist is only a heuristic, nevertheless we are convinced that the heuristic will only seldom fail.

In spec. 9.9 on the next page we see the transformation of spec. 9.3 on page 200, where two elements, ⟨“Chartres”⟩ and ⟨“Amiens”⟩, are defined inside video ⟨“French”⟩. In Vitruv_L we generate the new class Video1 extending class Video and defining the two elements Chartres and Amiens as intervals. Their temporal arrangement is found in the body of class Video1. The quantitative duration constraint of “Amiens” is applied in the binding of element Amiens of class instance French.

9.3.4. Media Composition

While transforming the media composition we have to address various topics, most of them deal with the body of the class modeling the current scene. The temporal arrangement, which is rather freely defined in Vitruv_N, has to be normalized to legal Vitruv_L expressions. If we use events, suitable event declarations have to be made. We discuss these issues now.

Specification 9.9 Media definitions with elements in Vitruv$_L$

```
class Video1 extends Video
  exports Chartres , Amiens
  let
    Chartres : Interval;
    Amiens : Interval
  body
    this.alpha starts Chartres.
    Chartres meets Amiens.
  end;
end;
// in some other class
let
  French : Video1;
body
... // the remainder of the specification
bindings
...
  object ... // binding of some class
    object French : Video1
      object Chartres : Interval ... end;
      object Amiens : Interval
        let length := triangle (26, 30, 34);
      end;
    end;
  end;
end;
```

9.3.4.1. Temporal Arrangement

The temporal arrangement defined in Vitruv$_N$ applies to scenes. Therefore, the media composition maps primarily to the body of the class modeling the current scene. The interval relationships found in Vitruv$_N$ have to be transformed to their canonical form in the strict INTERVAL RELATION INTERVAL syntax used in Vitruv$_L$. Compound relations are not declared explicitly in Vitruv$_N$ but this is required for Vitruv$_L$. For the translation of compound relations two strategies are possible:

1. The transformation is only possible if for all compound relations used in Vitruv$_N$ a respective definition of the relation in Vitruv$_L$ exists. The typical place for such definitions is the prelude.

2. The transformation generates for each unknown compound relation an empty definition in Vitruv$_L$. The specifier is obliged to check the generated specification and to fill the empty definitions.

We prefer the second alternative, because it is more user friendly in our opinion. But both alternatives can be realized in an implementation.

In spec. 9.10 on the next page we present the transformation of spec. 9.4 (see p. 202), both specifications use the former specification of video “French” or class Video1, respectively. The type of French is class Video1, defined in spec. 9.9 on the preceding page. The relation between media “present Amiens” and element “Amiens” in Vitruv$_N$ are realized by a relation between presentAmiens and French.Amiens, the latter representing the media element. The activity of a button means that the corresponding event is enabled. The event is pressed, according to the former specification of buttons in spec. 5.6 on page 71. This knowledge about buttons and their structures makes evident why the standard set of media types in Vitruv$_N$ is not completely open: in general the transformation process needs knowledge about the intended meaning of the media definitions. Consequently, adding a media type to Vitruv$_N$ requires also a change in the transformation implementation.

9.3.4.2. Interaction

For interaction purposes Vitruv$_N$ allows four different kinds of events. Buttons can be used immediately, since their event declaration is already done in the prelude. For the three mouse events, the situation is different, since they can be used for every media. Therefore, we apply a similar strategy as for media elements: if a mouse event is used for an instance of a media type, then we generate a new class for the media inheriting from the respective media type and add to this new class a mouse event. If the events is applied to a media element, then we generate also a new class, inheriting, however, from class Interval, which is normally used for modeling media elements.

Access to events with an ⟨if⟩-statement is translated to a selector, which has to be declared in the let-block of the class definition. The selector is somewhat degenerated

Specification 9.10 Media composition without events in Vitruv_L

```
let
  French : Video1;
  BellsIntro : Audio;
  ChartresButton : Button;
  presentAmiens : Audio;
  ChartresExplained : Audio
body
  this.alpha starts French;
  BellsIntro.length is DURATION.short;
  BellsIntro isStartedBy French;
  BellsIntro meets ChartresExplained;
  presentAmiens during French.Amiens;
  ChartresButton.pressed equals French.Chartres;
  ...
end;
```

since it features only one path of control, to which we translate the consequent of the ⟨if⟩-statement. Branching to another scene is realized as always in Vitruv_L by using leave for. This is shown in spec. 9.11 on the following page, which is the Vitruv_L transformation of spec. 9.5 on page 205.

Translating loops to Vitruv_L is now straightforward. The loop is already declared in Vitruv_N, and has to be declared in Vitruv_L as an element of the scene class as well. The body of the loop enjoys the same temporal arrangement constructions as the media composition of a scene, so there is nothing new here. The only important difference is that all media and media elements occurring in the body of the loop in Vitruv_N are required to be local elements of the loop in Vitruv_L. For proper Vitruv_N specifications this is no problem, since no relation between temporal entities inside a loop and those outside a loop are allowed. We do not provide an example since Vitruv_N specification 9.6 on page 206 is built after its Vitruv_L counterpart in spec. 5.7 on page 73.

9.4. Assessment of Vitruv_N

For assessing Vitruv_N we carried out experiments. Our aim was to check whether Vitruv_N specifications are understandable without training or detailed explanation such that different experiment subjects come to the same interpretation of the specification. Since we want to test the simple understandability of Vitruv_N, we decided that the subjects have to draw a bar chart, where each bar indicates a certain interval and its position throughout the scene. This is very similar to the bar chart provided in fig. 3.1 on page 31 and is an apparent and simple notation for (simple) concurrent

Specification 9.11 Branching to another scene in $\mathsf{Vitruv_L}$

```
let
  S : Selector;
  ChartresButton : Button;
body
  ...
  s selects (ChartresButton.pressed) with rules
    on (isPressed) do
      let ChartresInfo : Scene;
      body
        alpha meets leave for (ChartresInfo);
      end;
  end;
  ...
end;
```

activities.

The simple bar notation restricts us to pre-orchestrated specifications, just because branching and loops are not well suited for this notation. Using branching and loops in the experiment would require a more difficult notation for the interpretation of the specification. Their use increases the risk that erroneous interpretations are based on misunderstandings of both, $\mathsf{Vitruv_N}$ and that particular notation. In such a case the analysis of negative experiment results is also difficult, because it is unclear where the roots of the problem are.

The experiment was undertaken by three subjects, including an English native speaker without a computer science education. We gave the subjects a scene description (the intro scene from sec. 10.2.1 on page 220, in a first version of $\mathsf{Vitruv_N}$) together with a brief introduction of less than five minutes. Their task was to draw a bar chart of the temporal arrangement of the scene, showing position and length of the media and their elements. The results were promising, since all bar charts were very similar, differences result primarily from unspecified media durations. However, some derivations from the correct solution had other reasons: using Allen's relation directly does not fit to the semantics of every day English. This result was also discussed earlier in sec. 9.2.4.1 on page 201. A second experiment with the same scene, but the current version of $\mathsf{Vitruv_N}$, did not show these derivations.

Nevertheless, the discussion after the experiments indicated some disadvantages of $\mathsf{Vitruv_N}$. Concerning multimedia presentations, the users missed information of the spatial arrangement, required for building the mental image of the specified presentation. Of course, this is a disadvantage by design, since we explicitly omitted any spatial information from Vitruv, but focus on temporal specifications only.

A second problem mentioned in the discussion was that only binary relations be-

tween media are considered. This results sometimes in quite boring and repeating sentence structures. A typical example is a sequence of intervals, which are described by structures like A meets B, B meets C, C meets D etc. Another example is the parallel start of some intervals, e.g. in A starts B. A starts C. A starts D. Here, n-ary relations were preferred by the users, e.g. A starts B, C and D.

Concluding, we can say that the experiments have shown that the overall approach of Vitruv$_N$ looks promising.

Part III.

Applying Vitruv

10. The Multimedia Cathedral

In this chapter we present as a larger example a multimedia presentation specified in Vitruv. The example presents French Gothic cathedrals and is intended as an extension of the Altenberg Cathedral Presentation (sec. 1.1 on page 1). We present three scenes in detail, but make references to other scenes as well, which are required to specify a complete application: three scenes are not enough for all the beautiful cathedrals. Beside the ability to specify multimedia presentations with Vitruv, we show in this example how the different models of Vitruv are related and how the models cooperate in the transformation from the initial Vitruv_N specification to the formal semantics as Vitruvian Nets.

In sec. 10.1 we sketch the scenes very briefly, followed in the next sections with more elaborated specifications in Vitruv_N (sec. 10.2 on the next page), transformed to Vitruv_L in sec. 10.3 on page 224 and finally in sec. 10.4 on page 238 as Vitruvian Nets. In sec. 10.5 on page 248 we conclude the example with a summarizing discussion.

10.1. Scene Collection

In this section we describe the three scenes very briefly to guide the reader more easily through the example. In the setting of a development process, the description here can also be seen as a first coarse grained requirements statement, which is refined in the next sections.

Intro Scene

The introduction scene presents a video moving from a cathedral ground plan to the west work, through the rose window into the interior. Then the focus moves to the colored windows and the video finishes while showing a medieval scene in one of the windows.

During the video, we hear organ music, superimposed with bells during the west work shot and with medieval music parallel to the medieval scene. Additionally, the title and subtitle of the presentation is shown during the first parts of the video.

Main Menu

The main menu scene is the central scene of our presentation. Various buttons allow to branch to different subsequent scenes and to exit the presentation.

Various Cathedrals
This scene is well known from various examples throughout the thesis. We see some cathedrals, including those of Chartres and Amiens, in an animation. For each cathedral a button is shown, a click on the respective button leads to another scene explaining that cathedral in more detail.

10.2. Specifying with Vitruv_N

In this section we specify with Vitruv_N the scenes sketched in the last section. To ease reading the specification we have enhanced the layout of the specifications: headings are set in boldface and for enumeration lists of media and elements we use lists with bullets; comments within the specification use an italic font.

For each scene, we have a section of its own. The specifications found there have to be embedded into the following torso:

Beginning of scene definitions:

Description of scene "The Intro".
Details are in sec. 10.2.1.

Description of scene "Main Menu".
Details are in sec. 10.2.2 on the facing page.

Description of scene "Various Cathedrals".
Details are in sec. 10.2.3 on page 222.

Some more scenes may be specified here. In particular the scenes presenting details of the cathedrals of Amiens, Chartres, Paris, Reims and Rouen, which are referred in sec. 10.2.3 on page 222.

End.

10.2.1. The Intro

The introduction is a typical pre-orchestrated scene without any user interaction.

Description of scene "The Intro".

It is a short scene.

In this scene, these media are used:

- An image identified by "the Title".
- An image named as "Subtitle".
- A video named as "intro". In this media the following elements exist:
  - An element named "rose windows".

  - An element identified by “west work”.
  - An element identified by “ground plan”.
  - An element named “colored windows”.
  - An element named “medieval scene”.

  Element composition:

  Element “rose windows” is shown completely during “west work” and is about 2 seconds long. Element “west work” is overlapped a very little by “ground plan”. Element “ground plan” starts immediately after the beginning of the video. Element “colored windows” is overlapped slightly by “medieval scene” and is met by “west work”. Element “medieval scene” and video “intro” finish together.

  End of the content description.

- A short audio-clip identified by “bells”.
- An audio-clip named as “organ music”.
- A short audio-clip identified by “medieval music”.

End of the media definition.

The media composition:

The scene starts with video “intro”. Parallel to video “intro” we hear the audio-clip “organ music”. Presenting the image “the Title” overlaps largely video element “ground plan”. Shortly after image “the Title” the image “Subtitle” is shown. Video element “west work” and the rendering of image “Subtitle” finish together. During video element “west work” we hear audio-clip “bells”. Audio-clip “medieval music” sounds during the “medieval scene”. Immediately after video “intro”, we activate scene “Main Menu”.

End of the media composition.

End of the scene description.

10.2.2. Main Menu

The main menu contains a loop, which is finished by pressing the exit button. After the loop, nothing else happens and thus the presentation ends. Inside this loop, we activate button ⟨“The Cathedral Tour”⟩, leading to the respective scene. If we had additional menu entries, we would add a button for each entry. We make the entire loop as long as the audio-clip, which guarantees that the loop body finishes when the audio-clip ends. Hence, the audio-clip is repeated until the exit button is pressed.

Description of scene “Main Menu”.

In this scene, these media are used:

- A button identified by “Exit-Button”.
- A button named as “The Cathedral Tour”.
- A not too long audio-clip identified by “atmosphere”.
- A loop named “waiting for the end”.

End of the media definition.

The media composition:

The scene starts with activation of button “Exit-Button”. Loop “waiting for the end” repeats until button “Exit-Button” is pressed:

The loop starts with activating button “The Cathedral Tour”.

The loop is equal to the audio-clip “atmosphere”.

If button “The Cathedral Tour” is pressed then scene “Various Cathedrals” is activated.

End of the loop.

End of the media composition.

End of the scene description.

10.2.3. Various Cathedrals

This scene is similar to various examples throughout this thesis. We have a video and an audio-clip, the elements of both synchronously present various cathedrals. During the presentation of each cathedral, a button appears. A click to this button links to another scene (not shown here), which shall present the respective cathedral in more detail.

The very regular structure of the scene is also present in the specification. Since the sequence of the video and audio elements are equally structured, it is sufficient that we only specify the video element sequence in the element composition of the video, whereas the audio elements are arranged to be more or less equal to the video counterparts in the media composition part of the scene. An even more regular structure of the scene would include also the sequence of audio elements in the media definition part.

Description of scene “Various Cathedrals”.

In this scene, these media are used:

- A button identified by “To Chartres”.
- A button named as “To Amiens”.
- A button named as “To Reims”.

- A button identified by “To Paris”.
- A button named “To Rouen”.
- A 2 minute long video, named as “Some Cathedrals”. In this media the following content elements exist:
  - An element named as “Chartres”.
  - An element identified by “Amiens”.
  - An element named as “Reims”.
  - An element identified by “Paris”.
  - An element identified by “Rouen”.

  Element composition:

  The element “Chartres” starts immediately after the beginning of the video. The element “Amiens” is 30.0 seconds long and follows element “Chartres”. The element “Reims” is shown immediately before element “Paris” and has a duration of nearly 25 seconds. The element “Reims” finishes element “Amiens”. The element “Rouen” is about 25 seconds long and follows element “Paris”. The end of element “Rouen” is equal to the end of the video.

  End of the content description.
- An audio-clip identified as “Characterize the Cathedrals”. In this media, the following content element exists:
  - An element identified by “Explain Chartres”.
  - An element named as “Explain Amiens”.
  - An element named as “Explain Reims”.
  - An element named as “Explain Paris”.
  - An element identified by “Explain Rouen”.

End of the media definition.

The media composition:

The scene starts with video “Some Cathedrals”. Immediately after the video “Some Cathedrals”, we replace the current scene with scene “Main Menu”.

During video element “Chartres” we hear the audio element “Explain Chartres”. Parallel to video element “Chartres”, button “To Chartres” is active. If button “To Chartres” is pressed then we leave for scene “Chartres Details”.

The audio element “Explain Amiens” is played during video element “Amiens”. The Activation of button “To Amiens” is at the same time as the rendering of video element “Amiens”. If button “To Amiens” is pressed then scene “Amiens Details” is activated.

The video element “Reims” contains the audio element “Explain Reims”. Parallel to video element “Reims”, button “To Reims” is active. If button “To Reims” is pressed then scene “Reims Details” is activated.

During video element “Paris” we hear the audio element “Explain Paris”. Parallel to video element “Paris”, button “To Paris” is active. If button “To Paris” is pressed then scene “Paris Details” is activated.

During video element “Rouen” we hear the audio element “Explain Rouen”. Video element “Rouen” is shown at the same time as the activation of button “To Rouen”. If button “To Rouen” is pressed then the current scene is replaced with scene “Rouen Details”.

End of the media composition.

End of the scene description.

10.3. Cathedrals in Vitruv_L

We now present the scenes of our example presentation written in Vitruv_L.

We model the class structure of each scene as an UML diagram, using the convention that aggregation of predefined classes (i.e. of Interval, Video, Audio, Image, Text, and Button) is modeled as attribute declarations, whereas aggregation of classes defined in the scene is modeled by aggregation associations between the respective classes. Marker classes, such as Scene, are tagged with the stereotype <<Marker Class>>.

The following listings for the scenes are self-contained, i.e. we provide class, relation and type definitions together with their respective binding. This breaks the rather longish specification into three shorter parts, one for each scene. This eases the explanation of each scene. Of course, for a complete and syntactically correct specification, all three listings have to be merged as indicated in listing 10.1.

The transformation from Vitruv_N to Vitruv_L follows the rules given in sec. 9.3 on page 207. The entire Vitruv_N presentation is translated to a set of classes, definitions for compound relations and the respective binding. Each scene in Vitruv_N is mapped to a class inheriting from class Scene, where the first scene instantiation in the binding defines which scene is the starting scene in the presentation. Here, it is scene object introScene, representing scene ⟨“The Intro”⟩. We reference the remaining rules when they are applied for the first time in this example.

Listing 10.1: The Code-Frame for the Cathedrals Presentation

```
1   // We start with the definition of the Introduction Scene,
2   // shown in listing 10.2
3   define interval relation A overlapped a very little by B =
4     // ... and other compound relations
```

```
5       // and classes of Scene IntroScene
6     end; // of class IntroScene
7
8     // We go on to the definition of the Main-Menu Scene,
9     // shown in listing 10.3
10    class MainMenuScene
11      // ...
12    end; // of MainMenuScene
13
14    // Now to the definition of scene Various Cathedrals,
15    // shown in listing 10.4
16    class SomeCathedralsVideo
17      // ... and other classes of Scene VariousCathedralsScene
18    end; // of class VariousCathedralsScene
19
20
21    // Finally the combined binding section
22    bindings
23      object introScene : IntroScene
24        // The binding is shown in listing 10.2
25      end;
26      object MainMenu : MainMenueScene
27        // The binding is shown in listing 10.3
28      end;
29      object variousCathedrals : VariousCathedralsScene
30        // The binding is shown in listing 10.4
31      end;
32    end;
```

10.3.1. The Introduction Scene

The introduction scene ⟨"The Intro"⟩ consists of several media, but the only one of them with internal structure is video ⟨"intro"⟩. Following the rules for media definitions in sec. 9.3.3 on page 208, we have only to introduce a new class for the video ⟨intro⟩ inheriting from class Video, all other media are only instances of the predefined media classes. This structure is shown in the UML diagram in fig. 10.1 on the following page: classes TheIntro and introVideo represent the scene itself and the video ⟨"intro"⟩, respectively. In introVideo the elements are declared with type Interval (cf. sec. 9.3.3).

In listing 10.2 on page 227 we present the Vitruv_L specification of scene Intro, corresponding to the UML diagram in fig. 10.1 on the following page. The four compound relations used in scene ⟨"The Intro"⟩ are explicitly defined in the specification, according to sec. 9.3.4.1 on page 212. For the sake of completeness, we present them not only

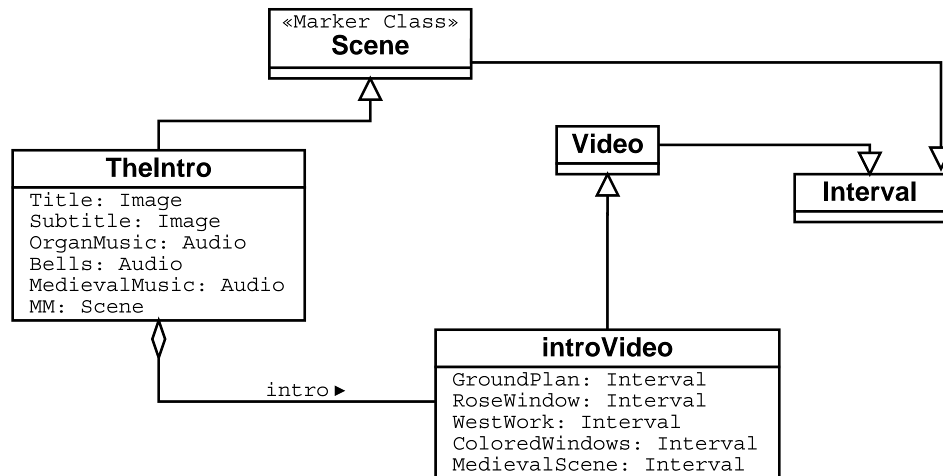

Figure 10.1.: UML Class Diagram for the Scene "Intro"

as empty definitions but add proper definitions. However, these definitions are not covered by the systematic translation of Vitruv_N to Vitruv_L, but are the task of the developers working on the transformation. To make the comprehension of the relation definition easier, we add to the definition a pictorial description of the temporal arrangement of the intervals used. The regular interval relationships, i.e. not compound ones, are transformed from Vitruv_N following the mapping presented in table 9.2 on page 203.

The final statement in scene ⟨"The Intro"⟩ activates scene ⟨"Main Menu"⟩. This is translated into a leave for statement in the class definition of IntroScene and requires in the binding a reference to an instance of class MainMenu (cf. sec. 9.3.4.2 on page 212).

Not all media or media elements in scene ⟨"The Intro"⟩ have an internal structure or duration assignments. Nevertheless, all media and media elements referenced in the Vitruv_N specification are required to be instantiated in the Vitruv_L specification. Therefore, following the rules in sec. 9.3.3 on page 208, we have several object-entries in the binding, which are empty (e.g. object OrganMusic : Audio end;), but we guarantee that the respective objects are instantiated.

For media or media elements with specified durations, we need assignments in the binding (cf. sec. 9.3.3). In the Vitruv_N specification (sec. 10.2.1 on page 220), the duration of ⟨rose windows⟩ was specified as about 2 seconds long. This requires an assignment of a fuzzy set, with 2 in its core, to the length of RoseWindow, the Vitruv_L counterpart of ⟨"rose windows"⟩. We decide here to use the triangular fuzzy set triangle (1.5, 2.0, 2.5), which models a duration of about 2 seconds. However, the decision for this particular fuzzy set is somewhat arbitrary. In general, the value used depends on the developers involved in the transformation process, or on an appropriate heuristic in an automatic translation process. However, if the initial value is not appropriate,

then, according to our proposal of a process model for Vitruv (sec. 4.5 on page 56), we can try a better value in the next iteration.

Listing 10.2: The Introduction Scene

```
1  // The Introduction Scene
2
3  define interval relation A overlapped a very little by B =
4    // BBBBBBBBBBBB
5    //          AAAAAAAAAAAAA
6    // CCCCCCCCCDDDEEEEEEEEEE
7    let C, D, E
8    in rules
9      A overlapped by B;
10     C starts B;
11     C meets A;
12     C meets D; D meets E;
13     D finishes B;
14     E finishes A;
15     D.length is extremly (short) ;
16   end;
17
18 define interval relation A overlapped largely by B =
19   //    AAAAAAAAAAAAA
20   // BBBBBBBBBBBB
21   // CCCDDDDDDDDDEEEE
22   let C, D, E
23   in rules
24     A overlapped by B;
25     C starts B;
26     C meets A;
27     C meets D; D meets E;
28     E finishes A;
29     D finishes B;
30     D.length is large;
31   end;
32
33 define interval relation A overlaps slightly B =
34   //           BBBBBBBBBBBB
35   // AAAAAAAAAAAAA
36   // CCCCCCCCCCDDDEEEEEEEEE
37   let C, D, E
38   in rules
39     A overlaps B;
40     C starts A;
```

```
41      C meets B;
42      C meets D;
43      D meets E;
44      E finishes B;
45      D finishes A;
46      D.length is slightly(short);
47    end;
48
49  class IntroScene
50    extends Scene
51    exports theTitle;
52    let
53      Bells : Audio;
54      intro : IntroVideo;
55      OrganMusic : Audio;
56      MedievalMusic : Audio;
57      theTitle : Image;
58      Subtitle : Image;
59      MM : Scene;
60    body
61      this isStartedBy intro;
62      intro equals OrganMusic;
63      theTitle overlapped largely by intro.GroundPlan;
64      Subtitle (shortly after) theTitle;
65      Subtitle (finishes or finished by) intro.GroundPlan;
66      Bells (starts or during or finishes) intro.WestWork;
67      MedevialMusic (starts or during or finishes) intro.MedievalScene;
68      intro meets leave for (MM);
69    end;
70  end;
71
72
73  class IntroVideo
74    extends Video
75    exports GroundPlan, RoseWindow, WestWork,
76            ColoredWindows, MedievalScene;
77    let
78      GroundPlan : Interval;
79      RoseWindow : Interval;
80      WestWork : Interval;
81      ColoredWindows : Interval;
82      MedievalScene : Interval;
83    body
84      this isStartedBy GroundPlan;
```

```
85      GroundPlan overlapped a very little by WestWork;
86      RoseWindow during WestWork;
87      ColoredWindows overlaps slightly MedievalScene;
88      MedievalScene finishes this;
89    end;
90  end;
91
92  bindings
93    object introScene : IntroScene
94      object Bells : Audio end;
95      object intro : IntroVideo
96        object GroundPlan : Interval end;
97        object RoseWindow : Interval
98          // rose window has a length of about 2 seconds
99          let length := triangle (1.5, 2.0, 2.5);
100       end;
101       object WestWork : Interval end;
102       object ColoredWindows : Interval end;
103       object MedievalScene : Interval end;
104     end;
105     object OrganMusic : Audio end;
106     object MedievalMusic : Audio end;
107     object theTitle : Image end;
108     object Subtitle : Image end;
109     let MM := ref (MainMenu);
110   end;
111   object MainMenu : MainMenuScene
112     // see listing 10.3 on page 231
113   end;
114 end;
```

10.3.2. The Main Menu

In scene ⟨Main Menu⟩ we have as declared entities a loop and two buttons, which handle user input. The transformation of the buttons follows those of other media, each button becomes an instance of the standard class Button. As reaction to button ⟨"The Cathedral Tour"⟩ we have an ⟨if⟩-statement in the media composition part of scene ⟨Main Menu⟩. This ⟨if⟩-statement and the loop are transformed into a Selector and a Loop instance, resp., including the access to the events of the two buttons, as defined in sec. 9.3.4.2 on page 212. The Loop instance is named in accordance to its Vitruv_N identifier as waitingForTheEnd, while we have to introduce a new identifier for the selector, because the selector has no corresponding named entity in Vitruv_N. We decided to use SelTCS as identifier for the selector. Since the ⟨if⟩-statement occurs

inside the loop body, the corresponding selector SelTCS is defined as local variable of the body of loop waitingForTheEnd.

The UML diagram in fig. 10.2 shows the class structure of the main menu scene, which is applied in listing 10.3 on the facing page. As usual, class MainMenuScene is derived from class Scene. Loop waitingForTheEnd and selector SelTCS inside the loop are modeled as entities of their own with stereotypes Loop and Selector, respectively. The graphical symbol is derived from UML's object icon. The aggregation relations show how these three entities are nested.

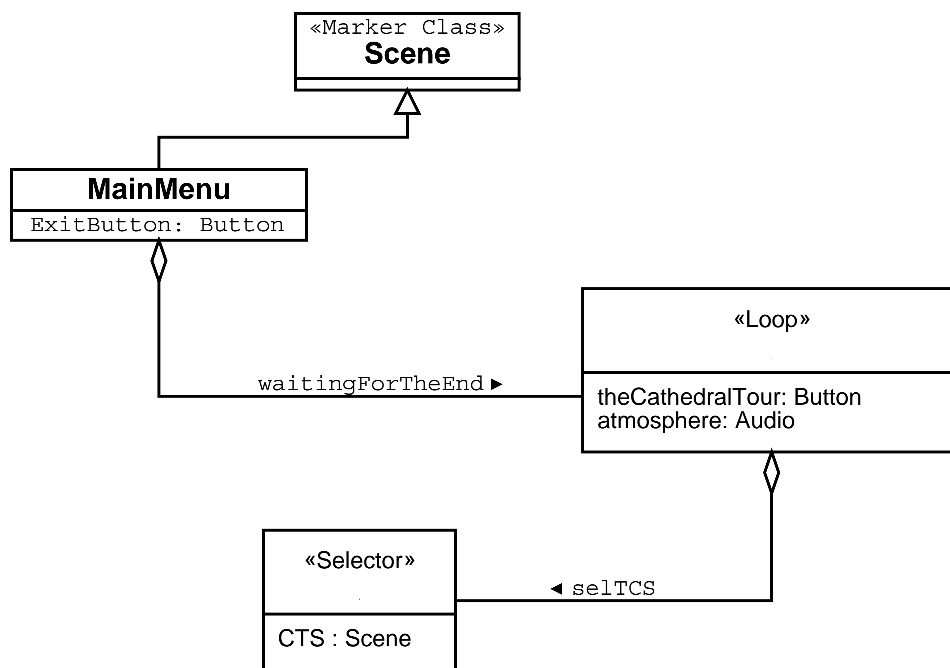

Figure 10.2.: UML Class Diagram for the Scene "Main Menu"

The constraint of audio-clip ⟨"atmosphere"⟩ to be ⟨not too long⟩ is modeled in the class definition of MainMenu as a qualitative constraint. In accordance to the definition in table 9.4 on page 204, we translate this constraint by applying modifier below on DURATION.long.

In the binding, we find instances of all used entities. Since we have no further quantitative information, buttons ExitButton and theCathedralTour and audio-clip atmosphere have empty bindings.

For the sake of completeness, we added in the binding a proper definition of the durations short and long of type DURATION exemplifying the use of context specific value assignments. We decided that a duration in the interval from zero to 45 seconds is short, and a duration greater than 2 minutes (120 sec.) can be considered as a long duration, a duration greater or equal than 3 minutes (180 sec.) is definitive a long du-

ration. Between the corner points (0 and 45, 120 and 180, resp.), the *s*- and *z*-functions, resp., define the possibility grades. But we have to remark that these values cannot be inferred from original Vitruv_N specification, since there is no indication what a long duration is. However, the values used show how knowledge about the application context can be applied for the translation from Vitruv_N to Vitruv_L.

Listing 10.3: The Main-Menu Scene

```
1  // Scene Main Menu
2
3  class MainMenuScene
4    extends Scene
5    let
6      ExitButton : Button;
7      waitingForTheEnd : Loop;
8    body
9      this isStartedBy ExitButton;
10     waitingForTheEnd loops
11       until (ExitButton.pressed.value is isPressed) do
12       let
13         theCathedralTour : Button;
14         atmosphere : Audio;
15         SelTCS :Selector;
16       body
17         this isStartedBy theCathedralTour;
18         this equals atmosphere;
19         atmosphere.length is below(DURATION.long) ;
20         SelTCS selects (theCathedralTour.pressed) with rules
21           on (isPressed) do
22             let CTS: Scene;
23             body
24               alpha meets leave for (CTS) ;
25             end;
26         end;
27       end;
28     end;
29   end;
30 end;
31
32 bindings
33   object MainMenu : MainMenuScene
34     object ExitButton : Button end;
35     in waitingForTheEnd : Loop
36       in type DURATION
37         let long := sfunction (120, 180) ;
```

```
38          let short := zfunction (0, 45);
39        end;
40        object theCathedralTour : Button end;
41        object atmosphere : Audio end;
42
43        in SelTCS : Selector
44          on
45            let CTS := ref (variousCathedrals);
46          end;
47        end;
48      end;
49    end;
50    object variousCathedrals : VariousCathedralsScene
51      // see listing 10.4
52    end;
53  end;
```

10.3.3. The Various Cathedrals

Scene VariousCathedrals has the most complex structure in our example, which is shown in the UML diagram in fig. 10.3 on the facing page, but the scene has a very regular structure though. Both media, video someCathedrals and audio-clip characterizeCathedrals, have an internal structure, which requires respective class definitions modeling the content elements. We have five buttons in the scene and for each button a selector branching to a distinct target scene.

In listing 10.4 on page 234, we present the Vitruv_L specification of scene VariousCathedrals, applying UML diagram 10.3 on the facing page. The temporal behavior is streamlined by the translation from Vitruv_N to Vitruv_L. Whereas the Vitruv_N specification applies a variety of expressions to make the sequence of sentences concerning the video and audio elements, buttons and branchings less monotonous, in Vitruv_L the similarity of these sections is stressed: the five sections in the body of class VariousCathedralsScene concerning the selectors are equal except for the identifiers used.

In class SomeCathedralsVideo we use relations where the order of arguments corresponds to the respective relations in the Vitruv_N specification (e.g. Reims meets Paris and Reims isMetBy Amiens corresponds ⟨“Reims” immediately before “Paris”⟩ and ⟨“Reims” finishes “Amiens”⟩, resp.). We do this for easier comprehension of the Vitruv_L code with respect to the Vitruv_N code. Alternatively, we could use the semantically identical uniform sequence of meets relations (here: Reims meets Paris and Amiens meets Reims), avoiding the inverse relation isMetBy.

As in the Vitruv_N specification, we have no relations between the elements of class CharacterizeCathedralsAudio, they are only used in the body of class VariousCathedralsScene, where they are related to the elements of class SomeCathedralsVideo. In

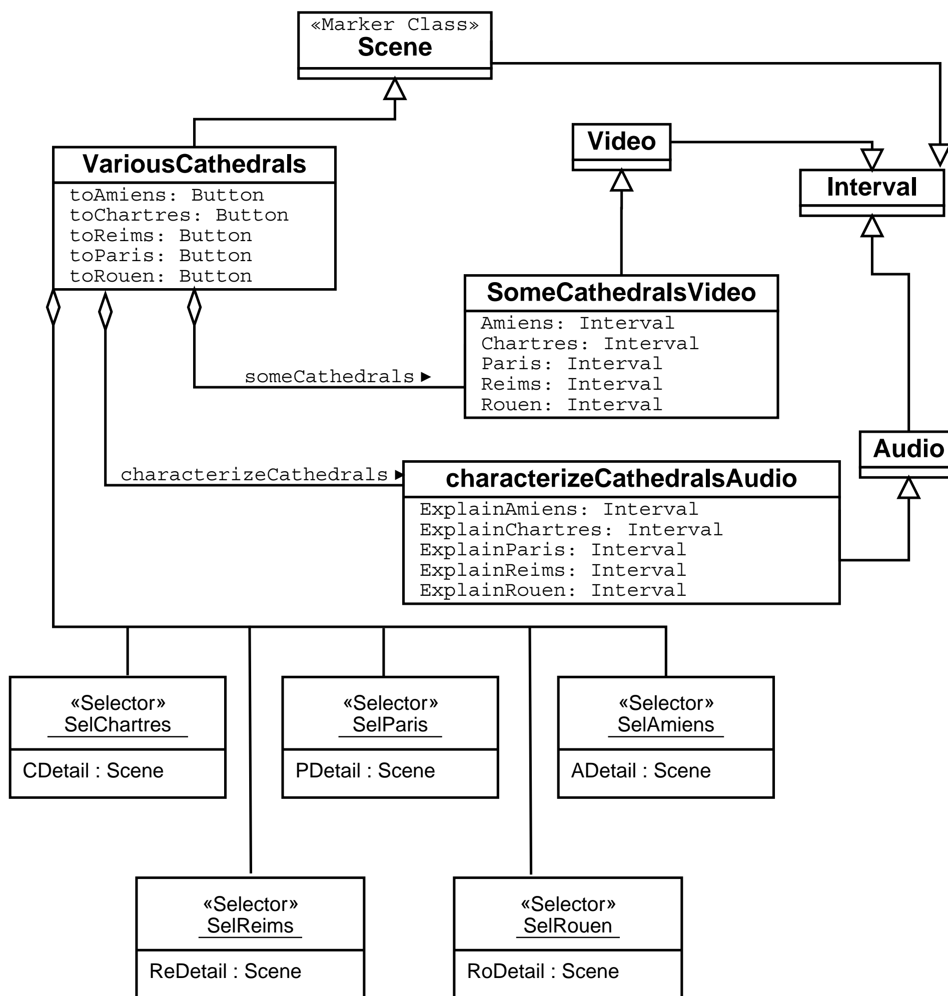

Figure 10.3.: UML Class Diagram for the Scene “Various Cathedral”

the binding, we have empty bindings for these elements, which guarantee that they are instantiated. All buttons are instantiated as well with empty bindings. All elements of object someCathedrals are also instantiated with empty bindings, except for the specified quantitative durations. We assign to elements Reims and Rouen triangular fuzzy sets, such that the duration of Reims has at most 25 seconds (triangle(22, 24, 25)) and that of Rouen has about 25 seconds (triangle(23, 25, 27)). Element Amiens is precisely 30 seconds longs, which is expressed by assigning a singleton fuzzy set, the fuzzy notation for a precise value.

Listing 10.4: Scene Various Cathedral

```
1  // Scene Various Cathedrals
2
3  class SomeCathedralsVideo
4    extends Video
5    exports Amiens, Chartres, Paris, Reims, Rouen;
6    let
7      Amiens : Interval;
8      Chartres : Interval;
9      Paris : Interval;
10     Reims : Interval;
11     Rouen : Interval;
12   body
13     Chartres starts this;
14     Amiens isMetBy Chartres;
15     Reims meets Paris;
16     Reims isMetBy Amiens;
17     Rouen isMetBy Paris;
18     Rouen finishes this;
19   end;
20 end;
21
22 class CharacterizeCathedralsAudio
23   extends Audio
24   exports ExplainAmiens, ExplainChartres, ExplainParis, ExplainReims,
25           ExplainRouen;
26   let
27     ExplainAmiens : Interval;
28     ExplainChartres : Interval;
29     ExplainParis : Interval;
30     ExplainReims : Interval;
31     ExplainRouen : Interval;
32   body
33     // no rules are defined for class CharacterizeTheCathedrals
34   end;
```

```
35 end;
36
37 class VariousCathedralsScene
38   extends Scene
39   let
40     toChartres : Button;
41     toAmiens : Button;
42     toReims : Button;
43     toParis : Button;
44     toRouen : Button;
45     SelChartres : Selector;
46     SelAmiens : Selector;
47     SelParis : Selector;
48     SelReims : Selector;
49     SelRouen : Selector;
50     someCathedrals : SomeCathedralsVideo;
51     characterizeCathedrals : CharacterizeCathedralsAudio;
52     MM : MainMenuScene;
53   body
54     this isStartedBy someCathedrals;
55     someCathedrals meets leave for (MM) ;
56
57     // the section for Chartres
58     characterizeCathedrals.ExplainChartres (starts or during or
59          finishes) someCathedrals.Chartres;
60     someCathedrals.Chartres equals toChartres;
61     SelChartres selects (toChartres.pressed) with rules
62         on (isPressed) do
63           let CDetail: Scene;
64           body
65             alpha meets leave for (CDetail) ;
66           end;
67     end;
68
69     // the section for Amiens
70     characterizeCathedrals.ExplainAmiens (starts or during or
71          finishes) someCathedrals.Amiens;
72     someCathedrals.Amiens equals toAmiens;
73     SelAmiens selects (toAmiens.pressed) with rules
74         on (isPressed) do
75           let ADetail: Scene;
76           body
77             alpha meets leave for (ADetail) ;
78           end;
```

```
79      end;
80
81      // the section for Paris
82      characterizeCathedrals.ExplainParis (starts or during or
83           finishes) someCathedrals.Paris;
84      someCathedrals.Paris equals toParis;
85      SelParis selects (toParis.pressed) with rules
86          on (isPressed) do
87            let PDetail: Scene;
88            body
89              alpha meets leave for (PDetail) ;
90            end;
91      end;
92
93      // the section for Reims
94      characterizeCathedrals.ExplainReims (starts or during or
95           finishes) someCathedrals.Reims;
96      someCathedrals.Reims equals toReims;
97      SelReims selects (toReims.pressed) with rules
98          on (isPressed) do
99            let ReDetail: Scene;
100           body
101             alpha meets leave for (ReDetail) ;
102           end;
103     end;
104
105     // the section for Rouen
106     characterizeCathedrals.ExplainRouen (starts or during or
107          finishes) someCathedrals.Rouen;
108     someCathedrals.Rouen equals toRouen;
109     SelRouen selects (toRouen.pressed) with rules
110         on (isPressed) do
111           let RoDetail: Scene;
112           body
113             alpha meets leave for (RoDetail) ;
114           end;
115     end;
116
117
118   end;
119 end;
120
121 bindings
122   object variousCathedrals : VariousCathedralsScene
```

```
123      object toChartres : Button end;
124      object toAmiens : Button end;
125      object toReims : Button end;
126      object toParis : Button end;
127      object toRouen : Button end;
128
129      object someCathedrals : SomeCathedralsVideo
130        object Amiens : Interval
131          let length := singleton (30) ;
132        end;
133        object Reims : Interval
134          let length := triangle (22 , 24 , 25) ;
135        end;
136        object Rouen : Interval
137          let length := triangle (23 , 25 , 27) ;
138        end;
139        object Paris : Interval end;
140        object Chartres : Interval end;
141      end;
142      object characterizeCathedrals : CharacterizeCathedralsAudio
143        object ExplainAmiens : Interval end;
144        object ExplainChartres : Interval end;
145        object ExplainParis : Interval end;
146        object ExplainReims : Interval end;
147        object ExplainRouen : Interval end;
148      end;
149      in SelChartres : Selector
150        on
151          let CDetail := ref (ChartresDetails) ;
152        end;
153      end;
154      in SelAmiens : Selector
155        on
156          let ADetail := ref (AmiensDetails) ;
157        end;
158      end;
159      in SelParis : Selector
160        on
161          let PDetail := ref (ParisDetails) ;
162        end;
163      end;
164      in SelReims : Selector
165        on
166          let ReDetail := ref (ReimsDetails) ;
```

```
167           end;
168         end;
169       in SelRouen : Selector
170           on
171             let RoDetail := ref (RouenDetails);
172           end;
173       end;
174       // now the local elements of variousCathedrals
175       let MM := ref (MainMenu);
176     end; // end of Scene variousCathedrals
177
178     object MainMenu : MainMenuScene
179       // see listing 10.3 on page 231
180     end;
181   // Following the scenes for the details of the various cathedrals
182 end;
```

10.4. The Cathedrals as Vitruvian Net

In this section we present our example presentation as a series of Vitruvian Nets. We have chosen a series of nets since the even the net for merely one scene get quickly large, such that it is too inconvenient to present one diagram only.

We start with the basic structure of the presentation in sec. 10.4.1, followed by a detailed look into scene MainMenu in sec. 10.4.2 on page 240. We have chosen this scene because it is small enough to be presented nicely and additionally it embodies all elements of Vitruv_L including selectors, loops, events and branching. The other two scenes have only subsets of these features: scene IntroScene consists of one block only and a final branching to MainMenu, scene VariousCathedrals does not feature a loop.

10.4.1. The Presentation Structure

The high-level view on our example presentation is shown in fig. 10.4 on the facing page and follows the construction defined in sec. 8.5.2 on page 181. In the center we have the place ConnectionPlace, connecting the three scenes IntroScene, MainMenu and VariousCathedrals. The level of detail in this figure corresponds to the Vitruv_N specification torso in sec. 10.2 on page 220 and to the Vitruv_L specification torso in spec. 10.1 on page 224, because we show only the existence of the three scenes. The Vitruvian Net construction for all the details found in the other Vitruv_N and Vitruv_L specifications is exemplified in sec. 10.4.2 on page 240 for scene MainMenu.

The three scenes are folded into three transitions for purposes of information hiding. Each scene is bracketed by respective start- and end-transitions, which connect

the scenes properly with place ConnectionPlace. The edges between scenes and ConnectionPlace have scene names inscriptions controlling which scene is started next. The initial marking would place a token with value IntroScene intro place ConnectionPlace.

From scene MainMenu we have two arcs from the scene to ConnectionPlace. The regular arc from endMainMenu transports a token with value s0. Since no scene with the name exists, the net is dead after endMainMenu has fired. This corresponds to the missing leave for in the body of class MainMenu: after the body is finished regularly (i.e. without premature branching), the presentation finishes as well. The second arc from scene MainMenu to ConnectionPlace models the branching from the menu to scene VariousCathedrals. The branching is done inside the menu and not parallel to the regular end of the menu, therefore we have a direct connection from a transition inside the scene to ConnectionPlace. The token moving along this arc has the value VariousCathedrals, hence scene VariousCathedrals is started next. In scene VariousCathedrals we have five branchings to scenes presenting various cathedrals (cf. spec. 10.4 on page 234). Since we omitted these scenes we also omitted the respective arcs from scene VariousCathedrals to place ConnectionPlace.

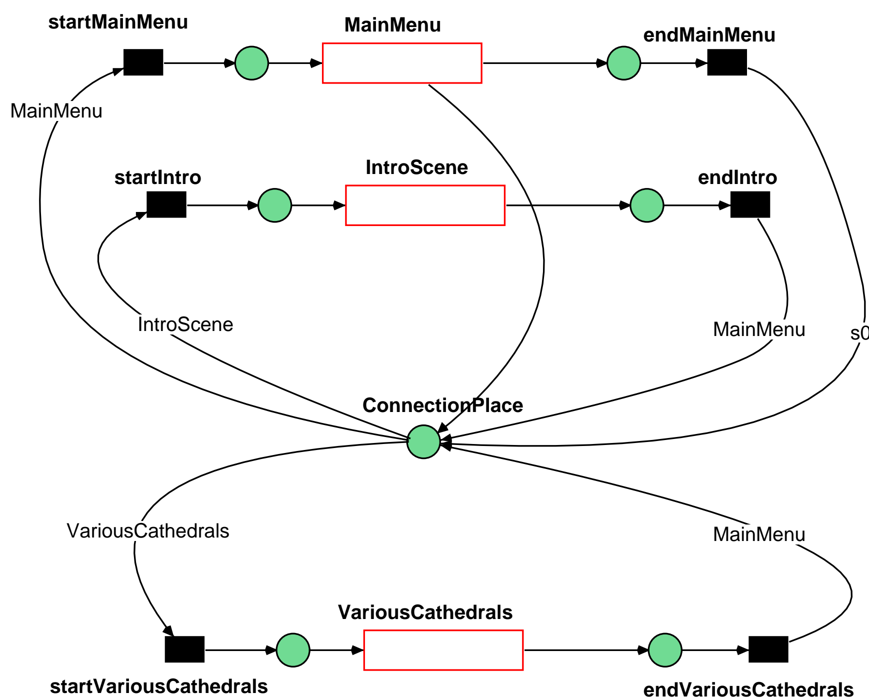

Figure 10.4.: The basic structure of the presentation

10.4.2. Scene MainMenu

An analysis of spec. 10.3 on page 231 reveals that the scene has three blocks:

1. the body of the scene,
2. the body of loop waitingForTheEnd and
3. the body of selector selTCS.

These three blocks are connected by the loop and the selector, respectively. Before we move towards the Vitruvian Net modeling scene MainMenu, we discuss the corresponding Vitruv$_l$ specification of the scene, since it is used for deriving parts of the Vitruvian Net.

10.4.2.1. Vitruv$_l$ Specification

In listing 10.5 on the next page we present the Vitruv$_l$ specification of scene MainMenu. We start with the binding of fuzzy type DURATION from the prelude (line 4), for the sake of brevity we restrict ourself to zero and below, which are used later. After that, the linearized binding of scene MainMenu follows (line 10). The linearization follows the *procedere* given in sec. 7.4 on page 142.

We start with the definitions for fuzzy type DURATION taken from the prelude, according to sec. 7.4.4 on page 146 and to sec. 7.4.1 on page 143. Since we need no further definitions from the prelude, we omit their representations in Vitruv$_l$.

The next part follows the rules given in sec. 7.4.3 on page 144: the linearized scene MainMenuScene follows the structure of the binding of instance MainMenu (compare spec. 10.3). We observe the nesting structure of the binding by the sequence of intertwined allocation and application phases: for flat objects, such as event ExitButton_-pressed of button ExitButton, their allocation and the application phases follow immediately, for nested objects, such as the root object MainMenu, the allocation phase (lines 14–21) and application phase (lines 124–130) bracket those phases of their inner objects.

In a Vitruv$_l$ specification, selectors and loops define blocks inside scenes. Following the rules given in sec. 7.4.5 on page 146, selector selTCS and loop waitingForTheEnd are simple intervals without internal structure and are separated from their respective bodies selTCS_body and waitingForTheEnd_body. Each body is the root interval of a corresponding block. Together with the block for the entire scene, we have three blocks: MainMenu_block, selTCS_body_block, and waitingForTheEnd_body_block.

In line 60 we show the re-binding of term long and short of fuzzy type DURATION. According to the rules in sec. 7.4.1 on page 143, we introduce new identifiers DURATION_long1 and DURATION_short1 and set their values. For their application during the translation process, we have to maintain properly the dictionary mapping Vitruv$_L$

identifiers to Vitruv_I identifiers: within the scope of loop waitingForTheEnd, DURATION.long is mapped to DURATION_long1, outside this scope we map DURATION.long to DURATION_long. The newly bound term long is used in line 87 and following, where we encode the Vitruv_L statement length is below (DURATION.long).

Listing 10.5: Scene MainMenu in Vitruv_I

```
1   /////////////////////////////////////////////////////////////
    // Defaults for DURATION, taken from the prelude
    /////////////////////////////////////////////////////////////
    modifier (DURATION_below) .
5   fuzzyvalue (DURATION_zero) .
    update (DURATION_below, below) .
    update (DURATION_zero, singleton (0.0) ) .

    /////////////////////////////////////////////////////////////
10  // Scene MainMenu
    /////////////////////////////////////////////////////////////

    // Allocation phase of Scene MainMenu
    interval (MainMenu) .
15  block (MainMenu_block) .
    inBlock (MainMenu, MainMenu_block) .
    fuzzyvalue (MainMenu_length) .
    element (MainMenu_length, MainMenu) .
    length (MainMenu, MainMenu_length) .
20  interval (MainMenu_alpha) .
    interval (MainMenu_omega) .
    // End of the allocation phase of MainMenu

    // Allocation phase of Button ExitButton
25  interval (ExitButton) .
    fuzzyvalue (ExitButton_length) .
    element (ExitButton_length, ExitButton) .
    length (ExitButton, ExitButton_length) .
    interval (ExitButton_alpha) .
30  interval (ExitButton_omega) .
    holds (MainMenu, ExitButton, { c, si, fi } ) .

    // Allocation phase for Event pressed of Button ExitButton
    // as usual
35  // Application phase for Event pressed of Button ExitButton
    holds (ExitButton_alpha, ExitButton_pressed, { s } ) .
    update (ExitButton_pressed_length, rectangle (0.0, inf) ) .
```

```
   // Application phase of Button ExitButton
40 holds (ExitButton, ExitButton_alpha, { si }).
   holds (ExitButton, ExitButton_omega, { fi }).
   constraint (ExitButton_alpha, value(DURATION_zero)).
   constraint (ExitButton_omega, value(DURATION_zero)).
   update (ExitButton_length, rectangle (0.0, inf)).
45
   // Allocation phase of loop waitingForTheEnd
   // as usual...

   // Allocation phase of the body of loop waitingForTheEnd
50 interval (waitingForTheEnd_body).
   block (waitingForTheEnd_body_block).
   inBlock (waitingForTheEnd_body, waitingForTheEnd_body_block).
   fuzzyvalue (waitingForTheEnd_body_length).
   element (waitingForTheEnd_body_length, waitingForTheEnd_body).
55 length (waitingForTheEnd_body, waitingForTheEnd_body_length).
   interval (waitingForTheEnd_body_alpha).
   interval (waitingForTheEnd_body_omega).

   // Type Duration inside loop waitingForTheEnd
60 fuzzyvalue (DURATION_long1).
   fuzzyvalue (DURATION_short1).
   update (DURATION_long1, sfunction (120, 180)).
   update (DURATION_short1, zfunction (0, 45)).

65 // Allocation phase of TheCathedralTour
   // as usual...

   // Allocation phase for Event pressed of Button TheCathedralTour
   // as usual...
70
   // Application phase for Event pressed of Button TheCathedralTour
   // as usual...


75 // Application phase of TheCathedralTour
   // as usual....

   // Allocation phase of atmosphere
   // as usual...
80 // Application phase of atmosphere
   holds (atmosphere, atmosphere_alpha, { si }).
   holds (atmosphere, atmosphere_omega, { fi }).
```

```
    constraint (atmosphere_alpha , value(DURATION_zero ) ) .
    constraint (atmosphere_omega , value(DURATION_zero ) ) .
85  update (atmosphere_length , rectangle (0.0 , inf ) ) .
    // use new duration term
    constraint (atmosphere_length ,
                apply (DURATION_below , value (DURATION_long1 ) ) ) .

90
    // Allocation phase of selector selTCS within loop waitingForTheEnd
    // as usual...

    // Allocation phase of the body of selector selTCS
95  interval (selTCS_body ) .
    block (selTCS_body_block ) .
    inBlock (selTCS_body , selTCS_body_block ) .
    fuzzyvalue (selTCS_body_length ) .
    element (selTCS_body_length , selTCS ) .
100 length (selTCS , selTCS_body_length ) .
    interval (selTCS_body_alpha ) .
    interval (selTCS_body_omega ) .
    holds (waitingForTheEnd_body , selTCS , { c , si , fi } ) .
    // Application phase of the body of selector selTCS
105 // as usual

    // Application phase of selector selTCS within loop waitingForTheEnd
    / as usual..

110 // Application phase of the body of loop waitingForTheEnd
    holds (waitingForTheEnd_body , waitingForTheEnd_body_alpha , { si } ) .
    holds (waitingForTheEnd_body , waitingForTheEnd_body_omega , { fi } ) .
    constraint (waitingForTheEnd_body_alpha , value(DURATION_zero ) ) .
    constraint (waitingForTheEnd_body_omega , value(DURATION_zero ) ) .
115 update (waitingForTheEnd_body_length , rectangle (0.0 , inf ) ) .
    // declared interval relations in waitingForTheEnd
    holds (waitingForTheEnd_body , theCathedralTour , { si } ) .
    holds (waitingForTheEnd_body , atmosphere , { = } ) .

120 // Application phase of loop waitingForTheEnd
    // as usual...

    // Application phase of Scene MainMenu
    holds (MainMenu , MainMenu_alpha , { si } ) .
125 holds (MainMenu , MainMenu_omega , { fi } ) .
    constraint (MainMenu_alpha , value(DURATION_zero ) ) .
```

```
    constraint (MainMenu_omega, value(DURATION_zero ) ) .
    update (MainMenu_length , rectangle (0.0 , inf ) ) .
    // declared interval relations in MainMenu
130 holds (MainMenu , ExitButton , { si } ) .
    // End of the application phase of MainMenu
```

10.4.2.2. The nets for the blocks

In the following we discuss the Vitruvian Nets modeling each block of scene MainMenu from the respective Vitruv_l specification (see listing 10.5 on page 241). In the Vitruv_l specification, we identified three blocks, the body of scene MainMenu, the body of loop waitingForTheEnd, and the body of selector selTCS. Before we discuss some details of these nets, let us explain their main structures in the next paragraph.

The body of scene MainMenu is shown in fig. 10.5 on the next page and contains primarily two parallel activities: event ExitButton_pressed is enabled and loop waitForTheEnd runs. Therefore, we fork the body into these two parallel paths shown on top of the net. The remainder of the net reflects the intervals and their relations as defined in the Vitruv_l specification, we observe a staircase pattern on the left and right side which is typical for sequences of starts and finishes relations. The body of loop waitingForTheEnd (fig. 10.6 on the facing page) has a similar structure, but we omit most alpha and omega intervals as explained later. In the loop body we have audioclip atmosphere parallel to the event pressed of button theCathedralTour, on which the selector selTCS depends. Finally, the body of selector selTCS (fig. 10.7 on page 246) shows the branching to scene VariousCathedrals as indicated by the triangular symbol.

In these three figures, we use the same convention as in sec. 8.2.2 on page 159, the black transitions are required for synchronization purposes and fire without any delay. All other transitions fire with fuzzy delays, but we omit the respective arc inscriptions for the sake of clearness. Additionally, transitions to be expanded have a yellow (or light gray) color (in fig. 10.5 on the next page transitions ExitButten_pressed and waitingForTheEnd), and transitions required for internal delays of interval relations have a pink (or dark gray) color (in fig. 10.5 on the facing page transitions delay_MM, delay_EB and delay_event).

In figures 10.6 on the next page and 10.7 on page 246, we omit the alpha- and omega-intervals of almost all intervals. Since all alpha- and omega-intervals have no specified relation in the Vitruv_L specification (with the exception of the body of the path in selector selTCS) and they have only a duration of zero, they appearance in the Petri nets would not add any further information and would only blow up the resulting nets. This blow-up can be observed when comparing figures 10.5 and 10.6: the former has 20 transitions, the latter only 13, but the respective block of loop waitingForThend has a more complex temporal structure than the body of scene MainMenu. The com-

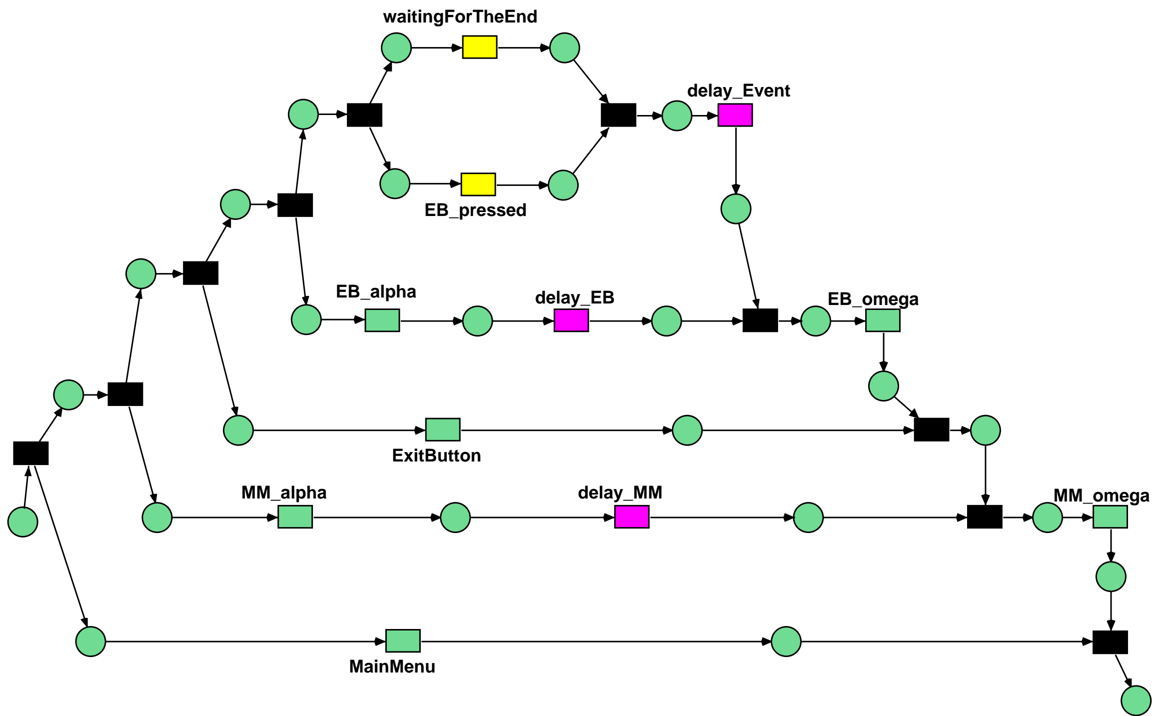

Figure 10.5.: The unexpanded body of scene MainMenu

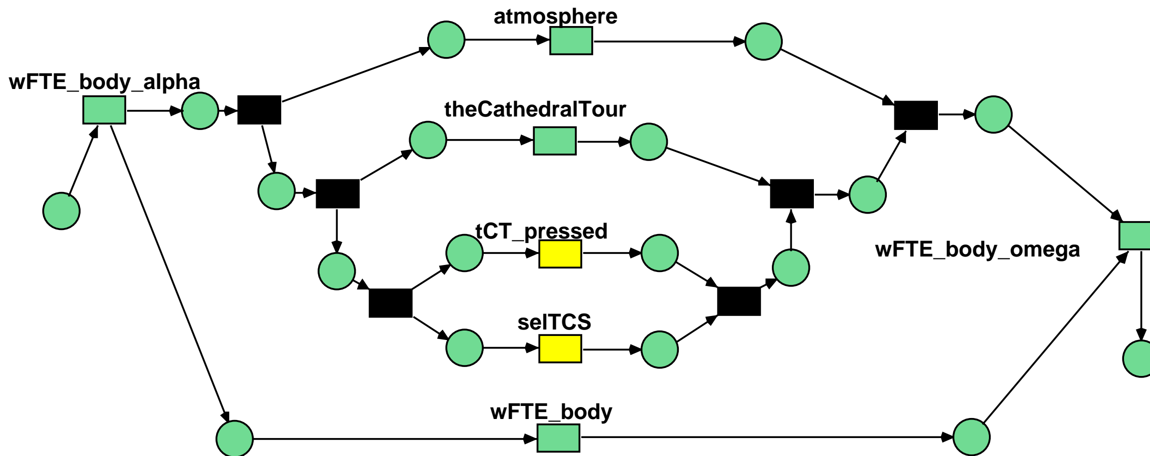

Figure 10.6.: The unexpanded body of loop waitingForTheEnd. We renamed the prefix waitingForTheEnd to wFTE and the event pressed of button theCathedral-Tour to tCT_pressed.

plete net for the loop body has 28 transitions: in addition the transitions of fig. 10.5 we have the interval atmosphere together with its alpha and omega intervals, the delay between alpha and omega, and four synchronization transitions. They realize the equals relation of atmosphere with wFTE_body, and the relations between atmosphere and its alpha and omega intervals.

In fig. 10.7 we use the triangle notation for the transition modeling the leave for statement as introduced in sec. 8.5.2 on page 181. We use this notation here for easier identification, but without its assigned semantics, since the proper translation of leave for statements is left to the final step of the algorithm for connection nets.

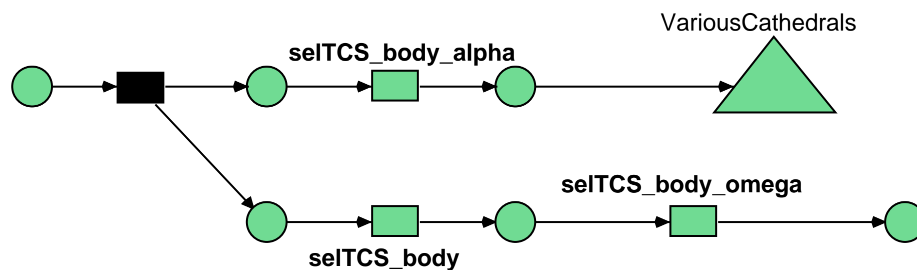

Figure 10.7.: The unexpanded path selTCS_body of selector selTCS

10.4.2.3. Connecting the nets

After preparing the nets for the blocks, we can apply the algorithm for connecting nets (fig. 8.10 on page 193). The first step is to start with the minimal regular Vitruvian Net, V, with its only place ConnectionPlace. We will only consider scene MainMenu, thereby the second step iterating over all scenes is executed only once. In step 2.a) we construct the basic Vitruvian Net of the block for scene MainMenu (shown in fig. 10.5 on the preceding page), and connect it with place ConnectionPlace. Now, we establish the list U of unexpanded events, loops and selectors of MainMenu, i.e. U contains event ExitButton_pressed and loop waitingForTheEnd.

In the next three steps (2.b) to 2.d)), we expand the elements of U. First, we generate for event ExitButton_pressed its event subnet E and replace the transition for ExitButton_pressed with E applying function *expandEvent* (def. 8.43 on page 188). The event is removed from U. Then, we generate the loop subnet L for loop waitingForTheEnd including the block of the loop body (fig. 10.6 on the preceding page) and replace the loop transition with L applying function *expandLoop* (def. 8.44 on page 189). This is the situation shown in fig. 10.8 on the facing page with the exception of the loop body, which we folded into transition body for the sake of brevity. The two expanded transitions are indicated by the two rectangles. The left rectangle comprises the net

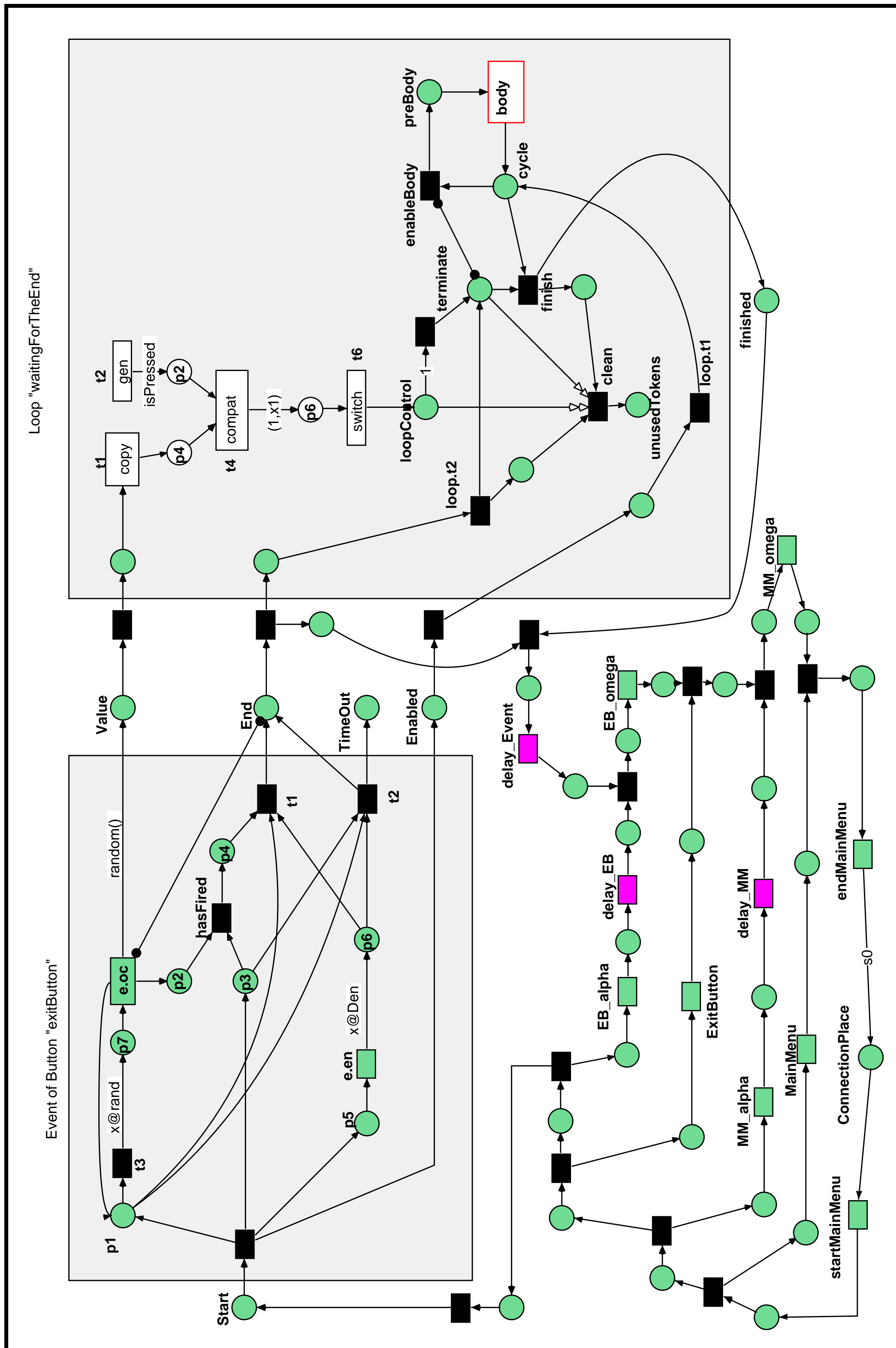

Figure 10.8.: The scene MainMenu without the loop body of waitingForTheEnd, which is shown in fig. 10.9 on page 249.

for event ExitButton_pressed (cf. sec. 8.4.2 on page 172), the right one shows loop waitingForTheEnd (cf. sec. 8.4.4 on page 176). For easier comparison, we reused the names of places and transitions from the formerly presented subnets. This is why several transitions with name, for instance, t1 exist. The connecting transitions between both rectangles are given in function *expandLoop*. Now we continue with the expansion. We delete waitingForTheEnd from U and add the unexpanded events, selectors and loops of the body of waitingForTheEnd to list U', i.e. we add event TheCathdralTour_pressed and selector selTCS to U'. Since we have no selectors in U, we omit the third step.

List U' is not empty, we have to expand its elements and repeat the last three steps after moving all elements from U' to U. First we replace event TheCathdralTour_pressed applying function *expandEvent*. The second step (2.c)) is omitted, since we have no unexpanded loop in U. In the third step, we expand selector selTCS including the net for the body of its path (see 10.7 on page 246) by applying function *expandSelector* (def. 8.45 on page 190). This situation is shown in fig. 10.9, where we focus on the body of loop waitingForTheEnd, since no modifications on the surrounding net has occurred. Again, we expand the event and the selector, and indicate this by the two rectangles. The left rectangle features the event of button theCathedralTour, the right rectangle shows selector selTCS (cf. sec. 8.4.3 on page 174). The selector is reduced to its one path, therefore the selector subnet differs from the net shown fig. 8.7 on page 175. In particular, the time-out value of the event is not used, however, we transfer control to particular place into the selector subnet. This is due to the application of function *expandSelector*, which takes no account of such possible optimizations.

Inside the body of the selector nothing has to be expanded, hence U' is empty, and we can leave the loop for expanding net elements. As we only consider scene MainMenu, step 2 is now finished, otherwise we would move to the next scene.

In the final step 3 of the algorithm, we add to the transition for the leave for statement inside the loop body an arc to place ConnectionPlace and reset arcs to all places of scene MainMenu. The algorithm terminates, the complete net (except for the other scenes) is constructed.

10.5. Summary

The example has shown how to specify a multimedia presentation with the Vitruv approach. We exemplified that we do not have to sacrifice the formal specification for the sake of common understandability, since we can derive the formal specification systematically from the Vitruv_N specification. With establishing the formal specification, formal analyses of the multimedia presentation can be done. This is the reason for our particular interest of the systematic transformation from the natural language specification into the formal models: first from Vitruv_N to Vitruv_L, then from Vitruv_L into its dynamic semantics, the Vitruvian Nets. The transformation process used in

Figure 10.9.: The Loop-Body of waitingForTheEnd. The triangular transition denotes the branching to scene VariousCathedrals.

this example followed the first steps of the process model sketch for Vitruv presented earlier in sec. 4.5 on page 56.

The example has also shown how the various models of Vitruv are related and how they cooperate. To illustrate the cooperation between the models of Vitruv let us discuss how the models deal with the intrinsic risks, viz ambiguity, imprecision, vagueness and incompleteness, which occur in Vitruv due to its NL basis. Taking care of these risks is not only needed when formalizing the Vitruv_N specification, but influences also the other models. Therefore, it emphasizes the model interaction.

Ambiguity is handled already in the definition of Vitruv_N by disallowing reflexive and pronoun language constructions, resulting in a superfluous but unambiguous use of definitive subject designators, e.g. in the media composition part of scene ⟨"Various Cathedrals"⟩ on page 223, where in each paragraph it is clear which button is meant but nevertheless the button's identifier is always required again. From a stylistic point of view concerning the use of the English language, this is a bit clumsy, but avoids misinterpretation, which is what we require here.

Vagueness and imprecision occur in the Vitruv_N specification in various situations, e.g. vague relationships such as ⟨overlaps largely⟩ (p. 221) or imprecise quantifications such as ⟨about 25 seconds⟩ (p. 223). They are handled in the formalization of Vitruv_N as compound relationships and fuzzy quantifications in Vitruv_L, where they can only occur in well defined places such that the link to the original specification in Vitruv_N is maintained. The valuation of fuzzy variables, required in the binding (e.g. in listing 10.4 on page 237 defining ⟨about 25⟩ as triangle (23, 25, 27)), is somewhat arbitrary, because there exists no counterpart in Vitruv_N specification. But the valuation honors the vagueness and imprecision found in the Vitruv_N specification, since we did not introduce any arbitrary nor overly precision just for the sake of the formalization.

Incompleteness indicates a lack of information. A typical situation in our example is the lack of quantitative information, found in both the Vitruv_N and the Vitruv_L specification. If we have additional information concerning the quantitative data, e.g. after an additional requirements elicitation, we can augment our model by adding these data to the specification, as discussed earlier in sec. 4.2 on page 52. For instance, we may add the missing durations of video elements Chartres and Paris of scene VariousCathedrals, modifying only the binding of Vitruv_L specification. If the new data does not contradict the remainder of the model, the data is regarded as valid. In this case, the qualitative model, i.e. anything except the binding, remains unchanged and stable. For Vitruvian Nets, however, the additional data would result in altered delay inscriptions, whereas the structure of the net remains stable as well. If the additional data excludes a former possible solution (e.g. we might reduce relation starts or during to starts), the net structure will become simpler, since we have to remove the part of the net representing the now excluded solution.

With this discussion we close our presentation of the Vitruv approach.

Part IV.

Summary and Future Work

11. Summary

In this thesis, we investigated a specification language for temporal aspects of multimedia presentations.

Starting from the experience in our own Altenberg Cathedral project and further analysis, we argued that in multimedia development new developer roles exist which we collectively call non-technical developers. The non-technical developers are distinguished from traditional technical developers by different educational backgrounds and skills, which are manifested prominently by a general dislike of formal notations. This situation makes it difficult to use traditional design and specification languages as a common basis for communication between all developers throughout the development process. This situation reminds us to the relationship between customers and developers during requirements engineering, where formal models are usually not applied for similar reasons. There, the formal models are derived from informal models by the virtuosity of the requirements engineers only. In our situation such an approach is not applicable, since we have to provide an adequate communication means between both groups throughout the entire development process.

In this thesis, we focus on the synchronization of media objects, i.e. we focus on the problem of how to specify the temporal arrangement of media objects. Technically, this is a challenging problem. Nevertheless, it is required for developing complex visualizations including user interaction but with the additional problem that both developer groups should be able to use the specification approach. Therefore, we stated four requirements for our approach, viz, a commonly understood (1) specification language (2) for synchronization of media objects (3), which shall be amenable for tool support (4). The literature reviews only the approaches discussed above, which satisfy some but not all of the requirements.

We identified natural language (NL) as a suitable common basis for communication between technical and non-technical developers. NL easily satisfies requirement 1, but has problems with imprecision, ambiguity, vagueness and incompleteness, which are considerable risks for using a NL-based specification language and for providing tool support. With the Vitruv approach, we have shown how to cope with these shortcomings.

We defined an understandable specification language (Vitruv_N) for temporal aspects of multimedia presentations, based on a subset of NL. This subset allows for tool support, since we define Vitruv_N with a (general) context free grammar, which eases parsing of specification documents. For semantical issues, we presented a formal counterpart, Vitruv_L, to Vitruv_N, to which a systematic transformation of Vitruv_N is

given. $\mathsf{Vitruv_L}$ is a statically typed specification language of its own, applying object-oriented concepts and fuzzy set theory for structuring specifications and for dealing properly with vagueness and imprecision inherent in the NL-based $\mathsf{Vitruv_N}$, respectively. For the event-free temporal structure, we use an extended variant of Allen's interval calculus. For event-based behavior, this is not applicable, and we developed a specialized Petri net model, Vitruvian Nets, to define the semantics of the event-based behavior of $\mathsf{Vitruv_L}$. Within the Vitruv approach, we consider $\mathsf{Vitruv_L}$ as mediator to the formal semantics of $\mathsf{Vitruv_N}$, i.e. the formal semantics of $\mathsf{Vitruv_N}$ are not given explicitly but implicitly as semantics of $\mathsf{Vitruv_L}$.

We have shown that it is possible to construct specification languages which are understandable to all participants of a development process, even in a complicated situation such as specifying dynamic system behavior. Even NL can be used, if we support its flexibility i.e. vagueness and imprecision, with respective formal models, as we have done with fuzzy set theory. The effect of incomplete specifications based on NL can be diminished in the Vitruv approach by reasoning and analyzing the formal counterparts, i.e. by studying the corresponding specifications in $\mathsf{Vitruv_L}$ and Vitruvian Nets. This is similar to traditional formal specification approaches, but the main difference in Vitruv is that we derive the formal specification systematically from the NL specification. Therefore, we are convinced that the formal specification meets the intended meaning of the informal specification.

Leaving the special domain of requirements engineering for multimedia, our results seem to promise that difficulties in requirements engineering, concerning understandability between different stakeholder groups, can be minimized in similar settings as used in the Vitruv approach.

12. Directions of Future Research

Due to the broad range of topics in this thesis, some aspects are not discussed exhaustively. In this chapter we present some aspects providing us with directions for future research. We identify future work in three areas, concerning the usability of Vitruv and tool support, language extensions, and finally generalizations.

12.1. Usability and Tools

In this thesis, we developed the foundations of the Vitruv approach. It is based on the observation that we require commonly understood specification approaches for temporal aspects of multimedia presentations in the presence of technical and non-technical developers.

Our experiences with non-technical developers in the Altenberg Cathedral Project have shown that it is very important to provide not only the rudimentary approach but to focus also on usability aspects. The experiments reported in sec. 9.4 on page 213 are a very first step to validate the vocabulary of Vitruv_N. Since these experiments focus only on Vitruv_N, they cannot address the entire Vitruv approach. Consequently, additional empirical research is required.

But, before doing that, we have to do further preliminary work. It is not sufficient to have an approach with a sound formal basis, we also need tools to ease the effective work and provide comfortable handling of Vitruv. The tools required include a sophisticated editor for Vitruv_N, a consistency checker for Vitruv_L and Vitruv_I, a Vitruvian Net generator and simulator, and compilers translating the different formalisms. For some of these tools, a first attempt has been undertaken.

In his diploma thesis, Christoph Begall (2002) is developing currently a consistency checker for Vitruv_L specifications, applying the theoretical results for constraint solving of qualitative and quantitative interval relationships combined with fuzzy set theory. The first task is to convert a Vitruv_L specification into Vitruv_I such that only flat structures of intervals, relationships and length constraints exist. The second job is to check the satisfiability of such structures and to deduce lowest upper and highest lower bounds for interval durations.

A first version of a simulator for Vitruvian Nets has been implemented (Störzel and Alfert, 2002), extending the simulation engine developed in the diploma thesis of Marc Störzel (2001). The simulator is based on Reference Nets (Kummer et al., 2001), which are high-level timed Petri nets with inscriptions in Java. The structure of Vit-

ruvian Nets is preserved within Reference Nets: we have only to provide fuzzy sets as color sets, the functions for fuzzy decision control (see sec. 8.3 on page 163), and have to deal properly with fuzzy delays. The fuzzy delays are used as sources for random variates such that the possibility distributions are interpreted as empirical frequency distributions (Neelamkavil, 1987, p. 126). We can choose, whether the values for these random variates are determined each time or only once for a simulation run. The latter variant is closer to the real world because the media durations are fixed throughout a presentation, the former variant emphasizes the vagueness of knowledge about the media durations. The random variates for event occurrence times and event values are independent of the considerations concerning the media durations: each time an event is activated the random variates are determined in accordance to their respective random distributions.

These tools are the first incarnation of an environment for applying Vitruv, needed for the aforementioned empirical research.

12.2. Language Extensions

The languages of Vitruv are designed to deal sufficiently with temporal aspects of multimedia presentations. There are, however, some areas where enhancements and extensions are possible and desirable.

We focus here on two interesting areas of enhancement for Vitruv_L with impact on Vitruv_L and Vitruvian Nets. First, a higher expressiveness of fuzzy types gives us better facilities for specifying with Vitruv_L. Second, the connection between scenes currently follows the link principle of hypermedia systems, but more sophisticated connection types are sometimes desirable.

As we show in the following discussion, these additional facilities have demanding requirements and modify Vitruv_L considerably. Thus, a careful analysis of the facilities' requirements and features is needed.

12.2.1. Fuzzy Types

Fuzzy types are an important part of Vitruv_L, but their expressiveness is somewhat limited. In this section we discuss two possible enhancements.

More General Universes for Fuzzy Types

Fuzzy types as defined in sec. 5.4 on page 67 can only have numeric or enumerated universes. From a type-theoretic point of view, these are very simple universes. Other type constructors, such as records, products, unions, etc., would be helpful to apply more ordinary data abstractions.

However, such data abstractions also require the means for handling them, in particular functional expressions and logical predicates. The introduction of such pred-

icates and expressions results in a powerful language of its own, the expressiveness of which may become similar to Z, VDM or algebraic specification approaches. Their existence would seriously change the characteristic of Vitruv_L, requiring completely different semantics.

Internal Modifiers and Functional Modifier Definition Language

The modifiers presented in table A.2 on page 274 are external modifiers. Additionally, in the literature we find internal modifiers (Thiele, 1998, p. 204) not changing the shape of the fuzzy set, but translating the fuzzy set along its universe:

$$\forall u \in \mathcal{U}: \quad f(X)(u) = (f \circ \mu_X)(u) = \mu_X(f(u))$$

But such modifiers do not fit in our framework, as the following example shows.

Let us consider the modifier very applied to a fuzzy set large, such that very just moves the fuzzy sets by an offset x_0 towards infinity. The resulting membership function would have the form

$$\mu_{\text{very large}}(x) = \mu_{\text{large}}(x - x_0)$$

This definition is problematic, because it would need either a second parameter defining offset x_0, or offset x_0 would be fixed in the definition. The latter one makes it impossible to reuse the modifier in different areas and application domains where the once fixed offset is not adequate.

Alternatively, we could use a higher-order functional sub-language for defining modifiers. Then we could apply e.g. currying to achieve unary functions from binary functions where the first argument is given and fixed. The primary modifier definition would be a higher order function taking only a (non-fuzzy-set) argument (e.g. the offset) and returning a function mapping a fuzzy set to another fuzzy set. This approach would allow us to use such internal modifiers because we can construct suitable modifier definitions from the primary modifier for each application domain. Additionally, such a language can provide means for user-defined modifiers, extending the currently restricted set of modifiers. Also, this language would possibly fit very well to more complex universes for fuzzy types, discussed above.

We expect that the proposed changes to fuzzy types and modifiers also would require major changes for Vitruv_I, at least such that the proposed sublanguage can be compiled to Vitruv_I. This requires support for functions, expressions and logical predicates in Vitruv_I as well.

12.2.2. Connections between Scenes

In Vitruv_L we have several abstractions of control flow. Classes, in particular with private elements, can be interpreted as macros similar to hierarchical Petri nets with transition substitution (Jensen, 1997). Loops and selectors model various control flow

alternatives. Scenes and links between them (realized by leave for) model the graph of the presentation's structure in the large.

However, more sophisticated connections between scenes are sometimes desirable. In particular connections with procedure-like semantics come to our mind. A prominent example is a help screen, the activation of which causes pausing the current scene, which is restored after leaving the help screen. Often such situations can be modeled inside classes, i.e. we do not switch to another scene but exchange only the media currently rendered.

An incorporation of procedure-like connections between scenes would require a fundamental modification of the semantics of Vitruv_L, because we have to deal with states, storing them on a run-time stack, and reloading them back from the stack. Replacing the stack with an unstructured state storage would simplify procedures to coroutines, but essentially the complexity remains. Procedures suggest parameters and recursion, adding the entire complexity of procedural or functional languages to Vitruv_L. In contrast to the considerations concerning the aforementioned sublanguages for fuzzy type, the procedural or functional language aspects appear at runtime and not only at compile time. This requires to model these features also in the dynamic semantics of Vitruv_L, which may be difficult for the current Petri net based dynamic semantics.

12.3. Generalizations

We can generalize our approach in two directions: first, to support other aspects of multimedia presentations, and second, to support other areas than multimedia.

12.3.1. Multimedia Complete

Intervals remain the main abstraction in Vitruv, if we broaden the capabilities of Vitruv towards multimedia presentations in general, since media objects are the centerpieces of multimedia presentation. Additional information about media objects such as denoting the corresponding media files, defining inscriptions on buttons, etc., can be realized as additional attributes of media objects in both, Vitruv_N and Vitruv_L. For the dynamic semantics, we can abstract from these information, since they deal only with temporal aspects. If we regard media files as implementation of media objects, we can use the additional attributes to check easily whether the implementation fits to the specification: the duration of the addressed media files have to match the specified duration of the respective media object. The exact durations found can be used for further constraint solving: we can then reduce the specified possibility distributions to singleton sets, adding more precision to the entire specification.

12.3.2. Beyond Multimedia

A different kind of generalization is to change the application domain of Vitruv. Timing considerations and dynamic behavior are important properties of any application, which have to be elicited and specified during the requirements engineering phase of software development.

As long as it is adequate to model these requirements in terms of intervals, their temporal relationships and durations, and finally events for input, the Vitruv approach can be used. Our experience with Vitruv suggests that customers and software engineers can work together, similar to Participative Design, applying Vitruv_N. The formalization to Vitruv_L and Vitruvian Nets gives software engineers the possibility to work with more formal models, which are, however, strongly related to the natural language model. In this way, the risk of misunderstandings and mistranslation during the transformation from one model into another could be reduced.

13. Final Remarks

With Vitruv we have presented an approach for dealing with vague and imprecise requirements, which occur as intrinsic properties of natural language formulations. We have applied fuzzy set theory to handle formally imprecision and vagueness.

In software engineering, fuzzy set theory and other approaches of soft computing are used for analyzing software and related systems, e.g. for automatically generating design models in reverse engineering or for predicting metrics and complexity measures from data sets of similar projects, as reported in the SCASE workshop (Jahnke and Ryan, 2001).

In this thesis, we applied fuzzy set theory in a different way. We do not use fuzzy set theory for creating models of the software, but applied fuzzy set theory within the models: in contrast to traditional analyzing approaches, we use fuzzy set theory in the synthesis part of software engineering coping with vague and imprecise requirements. We hope that our ideas help to improve modeling the customers' requirements in a more appropriate fashion.

Part V.

Appendices

A. Definition of Vitruv_L

In this chapter we define the concrete syntax of Vitruv_L and present the standard prelude.

A.1. Concrete Syntax

We define the concrete syntax of Vitruv_L in BNF starting with some preliminaries (notation, lexicographic definitions and basic building blocks). It is followed by the overall structure, fuzzy types, compound relations, events, classes, and finally the binding.

A.1.1. Preliminaries

A.1.1.1. Notation

We notate the grammar in a simple BNF form. Nonterminals are written in italics such as *Nonterminal*, terminals in sans-serif enclosed by single quotes such as 'terminal'. Alternatives are separated by |, the right-hand-side and the left-hand-side of the production are separated by ::=.

Lexicographic definitions defined by regular expressions. We apply the usual notation and use the operators defined in tab. A.1 on the next page where we assume that *a* and *b* are regular expressions. Again, terminals are written in single quotes.

A.1.1.2. Lexicographic Definitions

The production *Ident* defines an identifier, which is a usual alpha-numeric identifier: starting with a letter, followed by letters, digit or underscores in any order.

letter ::= ['a' - 'z', 'A' - 'Z']

digit ::= ['0' - '9']

Ident ::= *letter* (*letter* | *digit* | '_') *

We have two different kinds of numbers: real and integer number. As usual, for a real number we have to denote the point, otherwise it would be an integer number. A special symbol, inf, denotes the (positive) infinity for both, real and integer domains.

Operator	*Meaning*
ab	concatenation of *a* and *b*
$a*$	repeat *a* zero or more times
$a+$	repeat *a* one or more times
$a?$	matches *a* zero or one times
(a)	groups a regular expression
[ ]	introduce a character set
'A' – 'B'	range from 'A' to 'B' (only inside [ ])
$a\|b$	matches either *a* or *b*
.	matches any character
$	denotes end of line

Table A.1.: Operators of regular expressions

UnsignedReal ::= *digit*+ ('.' *digit*+ (['eE']['-+']? *digit*+)?)?

RealNumber ::= '-'? (*UnsignedReal* | 'inf')

UnsignedInteger ::= *digit*+

IntegerNumber ::= '-'? (*UnsignedInteger* | 'inf')

Comments in Vitruv$_L$ are similar to C++ and Java line comments. They start with '//' and everything until the end of the line is ignored.

Comment ::= '//' .*$

A.1.1.3. General Building Blocks

Some productions are used in many situations. We collect them in this section, because they have no proper home.

In declaration lists we often need a non-empty list of identifiers separated by commas:

IdentList ::= *Ident* | *IdentList* ',' *IdentList*

Compound relations are named by an identifier sequence separated only by whitespace. The identifier sequence may consist also of primitive relations.

IdentSeq ::= *Ident* | *PrimitiveRelation* | *IdentSeq IdentSeq*

Expressions in the rules of classes and compound relations use not only identifiers but also expressions of identifiers by applying the dot-notation. The reserved word 'this' applies to the current object.

IdentExpr ::= *Ident* | 'this' | *IdentExpr* '.' *Ident*

A.1.2. The Structure in General

A specification consists of three parts:

1. the prelude, defining a set of standard declarations,
2. the declaration of classes, relations and types,
3. the binding section, defining the values of the used variables and operators.

Specification ::= *Prelude Declarations Bindings*

Prelude ::= | 'prelude' *Declarations* 'end' ';'

Declarations ::= *ClassDec* | *FTypeDec* | *IntervalDec*
| *Declarations Declarations*

In the following we present these parts in greater detail.

A.1.3. Fuzzy Types

Fuzzy types consist of four parts, its name, a set of terms, an optional set of modifiers, and their universe. Generally, a fuzzy type defines a new value set together with typical values, the terms. These standard values can be modified by applying modifiers to terms.

FTypeDec ::= 'define' 'type' *Ident Universe DefineTerms DefineModifiers* 'end' ';'

DefineTerms ::= 'define' 'term' *IdentList* ';'

DefineModifiers ::= | 'define' 'modifier' *IdentList* ';'

The universe of a fuzzy type is either a crisp convex subset, i.e. an interval of $\mathbb{R}$ or $\mathbb{Z}$, or its possible values are explicitly enumerated identifiers.

Universe ::= 'universe' *RealInterval* ';'
| 'universe' *IntInterval* ';'
| 'universe' *EnumeratedSet* ';'

RealInterval ::= '[' *RealNumber* '..' *RealNumber* ']'

IntInterval ::= '[' *IntegerNumber* '..' *IntegerNumber* ']'

EnumeratedSet ::= '{' *IdentList* '}'

A.1.4. Compound Interval Relationships

Compound interval relationships modify Allen's thirteen predefined relationships. We define a compound relation as a kind of macro which is expanded if used in bodies of classes or in definitions of other compound relations. The rules are a subset of those used in class bodies, they are defined there (sec. A.1.6.1 on the facing page).

IntervalDec ::= 'define' 'interval' 'relation' *Ident CompoundRelation Ident* '='
LetIntervals IntRules ';'

CompoundRelation ::= *IdentSeq*

LetIntervals ::= 'let' *IdentList* 'in'

IntRules ::= 'rules' *IntRule* 'end'

IntRule ::= *IntervalRelationship* | *Constraint* | *IntRule IntRule*

A.1.5. Events

Events are declared to define the type of their value element.

EventDec ::= 'Event' '[' *Ident* ']'

A.1.6. Classes

Each class has a name and consists of four different parts discussed next. The inheritance declaration is optional, but a class has to export some of its elements. The local declaration define the elements of the class with their types, they are used in the class body.

ClassDec ::= 'class' *Ident Inherits Exports LocalDec Body* 'end' ';'

Inherits ::= | 'extends' *Ident*

Exports ::= | 'exports' *IdentList* ';'

LocalDec ::= 'let' *ElementDec*

ElementDec ::= | *Ident* ':' *ElementType* ';' | *ElementDec ElementDec*

ElementType ::= *Ident* | *EventDec* | 'Selector' | 'Loop'

Body ::= 'body' *Rule* 'end' ';'

A.1.6.1. Rules

The rules in class bodies and also the subset used in the definition of compound relations define the behavior.

Rule ::= | *Constraint* | *IntervalRelationship* | *UseLoop*
| *UseSelector* | *Rule Rule*

Constraints are fuzzy set expressions built on terms and modifiers of fuzzy types and the general binary operators or a reference to another element. As usual, and-operators have a higher priority than or. The productions *Term* and *Unary-Op* refer to terms and modifiers of the respective fuzzy types.

Constraint ::= *IdentExpr* 'is' *ConstraintExpr* ';'

ConstraintExpr ::= *ConstraintTerm* | *ConstraintExpr Or-Op ConstraintTerm*

ConstraintTerm ::= *UnaryExpr* | *ConstraintTerm And-Op UnaryExpr*

UnaryExpr ::= *ConstraintPrimary* | *Unary-Op* '(' *ConstraintExpr* ')'

ConstraintPrimary ::= *Term* | '(' *ConstraintExpr* ')'

Or-Op ::= 'or' | 'union'

And-Op ::= 'and' | 'intersect'

Unary-Op ::= *Ident*

Term ::= *IdentExpr*

Interval Relationships are surrounded by intervals which are either identifier expressions denoting some intervals or the leave for expressions denoting links to other scenes. Such relationship-constraints between two intervals can be a disjunction of different relationships as stated in sec. 3.1.2.2 on page 30 and following. Therefore, we can combine several compound relationships by disjunction.

IntervalRelationship ::= *Interval Relationship Interval* ';'

Interval ::= *IdentExpr* | *LeaveFor*

LeaveFor ::= 'leave' 'for' '(' *IdentExpr* ')'

Relationship ::= *CompoundRelation* | *PrimitiveRelation*
| *Relationship* 'or' *Relationship* | '(' *Relationship* ')'

$PrimitiveRelation$ ::= 'starts' | 'isStartedBy' | 'finishes' | 'isFinishedBy'
| 'meets' | 'isMetBy' | 'equals' | 'after' | 'before'
| 'overlaps' | 'isOverlappedBy' | 'during' | 'contains'

Selectors react on Events and decide which branch has to be activated. Each condition, the antecedent, is a fuzzy expression following the structure of a constraint defined above stating whether the event value is compatible to the constraint expression. Each branch, the consequent, has its own local elements and body following the structure of classes.

$UseSelector$ ::= $Ident$ 'selects' '(' $IdentExpr$ ')' 'with' 'rules' $SelectorRule$ 'end' ';'

$SelectorRule$ ::= 'on' '(' $SelectorAntecedent$ ')' 'do' $SelectorConsequence$
| $SelectorRule$ $SelectorRule$

$SelectorAntecedent$ ::= $ConstraintExpr$

$SelectorConsequence$ ::= $LocalDec$ $Body$

Loops define – similar to selectors – a body and local elements of their own and repeat the body until their event has value fulfilling the termination condition. The structure is very similar to selectors except that we have only one body.

$UseLoop$ ::= $Ident$ 'loops' 'until' '(' $IdentExpr$ 'is' $ConstraintExpr$ ')' 'do' $LocalDec$ $Body$
'end' ';'

A.1.7. The Binding

The purpose of the binding section is to assign explicit values to fuzzy sets and functions to modifiers. The structure of the prelude and the class definitions is repeated and a value or function is assigned to each definition either by inheriting a previous assignment or by a new one.

A.1.7.1. The Binding's Structure

The structure of the binding section follows the structure of prelude and class definitions, resp. If the prelude part is omitted, then the standard definitions will apply. The first object bound in production *SpecBindings* is the main entry point of the system specified.

$Bindings$ ::= 'bindings' $PreludeBindings$ $SpecBindings$ 'end' ';'

$PreludeBindings$ ::= | 'in' 'prelude' $PBind$ 'end' ';'

PBind ::= *InClass* | *InType* | *PBind PBind*

SpecBindings ::= *InObject* | *SpecBindings SpecBindings*

A.1.7.2. Binding of Fuzzy Types

Following the structure of a fuzzy type, we have to consider the terms and the modifiers of the type. The universe of the set is taken from the type's declaration. Values for modifiers are taken from the special object Modifier, hence production *VAssign* can handle all kinds of bindings in a fuzzy type.

InType ::= 'in' 'type' *Ident VAssign* 'end' ';'

A.1.7.3. Binding of Objects and Classes

The binding of an object is recursive, because we need also to bind all objects of which our object of interest consists. As each binding of an object opens a new context, we can also re-bind formerly defined fuzzy types. The last part of an object's binding is to bind the inherited private elements, that is done in production *InSuper*, moving up the inheritance hierarchy. The binding of classes in the prelude is similar to those of objects but starts with class.

InClass ::= 'class <Ident> <OAssign> 'end' ';'

InObject ::= 'object' *Ident* ':' *Ident OAssign* 'end' ';'

inSelector ::= 'in' *Ident* ':' 'Selector' *Path* 'end' ';'

inLoop ::= 'in' *Ident* ':' 'Loop' *OAssign* 'end' ';'

Path ::= *Path Path* | 'on' *OAssign* 'end' ';'

OAssign ::= *RebindTypes BindObjects ValueAssign InSuper*

ValueAssign ::= | *VAssign*

RebindTypes ::= | *InType RebindTypes*

BindObjects ::= | *BindObjects BindObjects* | *InObject* | *inLoop* | *inSelector*

InSuper ::= | 'in' 'super' *VAssign InSuper* 'end' ';'

A.1.7.4. Assigning of Values

All values in both, fuzzy types and classes, are bound in the same manner. Possible values are either fuzzy sets, other elements in the scope including special object Modifier for modifier definitions and references to a scene by ref.

$VAssign$ ::= 'let' $Ident$ ':=' $Value$ ';' | $VAssign\ VAssign$

$Value$ ::= $FuzzySetConstructor$ | $IdentExpr$ | 'ref' '(' $Ident$ ')'

Fuzzy sets can either be enumerated or constructed by convex or piece-wise linear constructors. For the enumerated fuzzy sets we need for each identifier its membership value. The convex constructors only need their respective sensitive points, whereas the generic constructor needs pairs of both, numbers and their membership values.

$FuzzySetConstructor$ ::= $FuzzySetEnumeration$ | $ConvexConstruction$
| $GenericConstruction$

$FuzzySetEnumeration$ ::= '{' $EnumValues$ '}'

$EnumValues$ ::= '(' $Ident$ ',' $RealNumber$ ')' | $EnumValues$ ',' $EnumValues$

$GenericConstruction$ ::= 'linear' '(' $MList$ ')'

$MList$ ::= '(' $Number$ ',' $RealNumber$ ')' | $MList$ ',' $MList$

$ConvexConstruction$ ::= $FuncOne$ | $FuncTwo$ | $FuncThree$ | $FuncFour$

$FuncOne$ ::= 'singleton' '(' $Number$ ')'

$FuncTwo$ ::= $FuncTwoIdent$ '(' $Number$ ',' $Number$ ')'

$FuncThree$ ::= $FuncThreeIdent$ '(' $Number$ ',' $Number$ ',' $Number$ ')'

$FuncFour$ ::= $FuncFourIdent$ '(' $Number$ ',' $Number$ ',' $Number$ ',' $Number$ ')'

$FuncTwoIdent$ ::= 'rectangle' | 'sfunction' | 'zfunction'

$FuncThreeIdent$ ::= 'triangle' | 'pifunction'

$FuncFourIdent$ ::= 'trapezoid' | 'szfunction'

$Number$ ::= $IntegerNumber$ | $RealNumber$

A.2. Standard Modifiers

Vitruv$_L$ provides a set of predefined non-domain-specific general modifiers shown in table A.2 on the following page. As general modifiers, they are external modifiers (Thiele, 1998, p. 204, from German: äußere), that means that the applications of modifier f to fuzzy set X with membership function μ and universe $\mathcal{U}$ results in

$$\forall u \in \mathcal{U}: \quad f(X)(u) = (\mu_X \circ f)(u) = f(\mu_X(u)).$$

These standard modifiers are adopted from the NRC Fuzzy Package (Orchard, 1999) and are based on various widely used definitions (Biewer, 1997). All standard modifiers can be applied to every fuzzy set, except above and below which need a numerical universe. Modifiers above and below realize the fuzzy sets $]X, +\infty[$ and $]-\infty, X[$, resp., defined in (B.43) and (B.45). We should explain the modifier norm, which normalizes its argument X (defined as above) and works in two steps. Firstly it determines the peak x_0 of X:

$$x_0 = \sup_{u \in \mathcal{U}} \mu(u).$$

Secondly, it divides each value of μ by x_0:

$$\forall u \in \mathcal{U}: \mathsf{norm}(X)(u) = \frac{\mu(u)}{x_0}$$

The effect is that each normal fuzzy set remains unchanged, because $x_0 = 1$, and otherwise each subnormal fuzzy set is raised, because $0 < x_0 < 1$: all elements with membership value x_0 become elements of the now non-empty core; the support remains unchanged. Normalization of the empty set is explicitly defined as the fuzzy set with membership function $\mu(x) = 1$.

A.3. The Standard Prelude of Vitruv$_L$

In listing A.1 on the next page we present the standard prelude of Vitruv$_L$. It contains a set basic definitions used for Vitruv$_L$, including fuzzy types, classes, compound relations and their default binding.

We start with four fuzzy types, namely the general types TRUTHand DURATION, and the enumerated types for events (ButtonState and MouseState). The root class Interval of Vitruv$_L$ is defined next. The classes Button and SensitiveArea provide events for buttons and mouse actions, respectively. These events are immediately enabled, any further constraints have to defined in derived classes. The last class in the prelude is the marker class Scene, which does not provide any new elements, but marks a derived class as a scene. Finally before the binding, we have the definition of the two compound relations shortly after and shortly before.

ButtonState
MouseState
Button
SensitiveArea

modifier name	*definition*
	$1 - \mu(u)$
	$\mu(u)/x_0$
	$(\mu(u))^2$
	$\sqrt[3]{\mu(u)}$
	$\sqrt{\mu(u)}$
	$(\mu(u))^{1.25}$
	$(\mu(u))^3$
	$\begin{cases} 2(\mu(u))^2 & \text{if } 0 \leq \mu(u) \leq 0.5 \\ 1 - 2(1 - \mu(u))^2 & \text{if } 0.5 < \mu(u) \leq 1 \end{cases}$
	intensify norm (plus $\mu(u)$ intersect not very $\mu(u)$)
	$]X, +\infty[$
	$]-\infty, X[$

Table A.2.: The standard modifiers applied to fuzzy set X given by its membership function μ with u ranging through the universe of X.

In the binding, we provide default values for the fuzzy types and class Interval. Neither the remaining classes nor the compound relations require additional values. For all fuzzy types, we do not redefine the modifiers, but stick to their default definitions. The terms of type TRUTHprovide fuzzy true and false values and additionally the values undefined and unknown, the membership function of which are defined as to be constant zero and one, respectively. For the terms of fuzzy types ButtonState and MouseState we define enumerated sets, which are only singletons. A button is pressed if the button state is down. For mouse events clicks and double-clicks can only appear, if the mouse is over the respective sensitive area.

The default values durations inside intervals are given in the binding of class Interval. The intervals alpha and omega have only a zero duration which is notated as a singleton fuzzy set at duration 0.0. On the other hand, the duration of an interval can be anything between zero and infinity, therefore all these durations are equally possible, notated as the rectangular fuzzy set from 0.0 to $+\infty$.

Listing A.1: The Standard Prelude of VitruvL

```
prelude

  define type TRUTH
    universe [0.0 .. 1.0] ;
    define term false , true , unknown , undefined;
```

```
    define modifier not, very, more_or_less;
  end;

define type DURATION
    universe [0.0 .. inf];
    define term zero, short, long;
    define modifier not, very, extremely, more_or_less, slightly;
end;

// For Button-Events
define type ButtonState
  universe { up, down };
  define term isPressed;
  define modifier not;
end;

// For Mouse-Events
define type MouseState
  universe { over, click, doubleclick };
  define term isRollover, isClicked, isDoubeClicked;
  define modifier not;
end;


// The root class of the inheritance hierarchy
class Interval
  exports alpha, omega, length;
  let
      alpha : Interval;
      omega : Interval;
      length : DURATION;
  body
      alpha (before or meets) omega;
      alpha is DURATION.zero;
      omega is DURATION.zero;
  end;
end;

// A button can be pressed.
class Button
    exports pressed;
    let
      pressed : Event[ButtonState];
    body
```

```
      this.alpha starts pressed; // only the enabling interval!
    end;
  end;

  // A SensitveArea is sensitve to mouse movements and clicks
  class SensitiveArea
    exports rollOver, clicked, doubleClicked;
    let
      rollOver : Event[MouseState];
      clicked : Event [MouseState];
      doubleclicked : Event[MouseState];
    body
      this.alpha starts rollover;
      this.alpha starts clicked;
      this.alpha starts doubleclicked;
    end;
  end;

  // Marker class Scene
  class Scene
    let
    body
    end;
  end;


  // Some compound relations
  define interval relation A shortly after B =
    // BBBBBBBBCCCAAAAAAA
    let C in rules
      B meets C;
      C meets A;
      C is short;
    end;

  define interval relation A shortly before B =
    // AAAAAAACCCBBBBBBBB
    let C in rules
      A meets C;
      C meets B;
      C is short;
    end;

end;
```

```
bindings
  in prelude
    in type TRUTH
      let false := zfunction (0.0, 0.3);
      let true  := sfunction (0.7, 1.0);
      let undefined := linear ( (0.0, 0) , (1.0, 0) ); // undef(x) = 0
      let unknown := rectangle (0.0, 1.0);  // unknwon (x) = 1
    end;
    in type DURATION
      let zero  := singleton (0.0);
      let short := zfunction (60.0, 120.0);
      let long  := sfunction (180.0, 240.0);
    end;
    in type ButtonState
      let isPressed := { (down, 1.0) , (pressed, 0.0) };
    end;
    in type MouseState
      let isRollover := { (over, 1.0) , (clicked, 0.0) ,
                          (doubleclicked, 0.0) };
      let isClicked  := { (over, 1.0) , (clicked, 1.0) ,
                          (doubleclicked, 0.0) };
      let isDoubleClicked  := { (over, 1.0) , (clicked, 0.0) ,
                                (doubleclicked, 1.0) };
    end;
    class Interval
      // alpha and omega have zero length
      object alpha : Interval
        let length := singleton (0.0);
      end;
      object omega : Interval
        let length := singleton (0.0);
      end;
      // each interval may have an arbitrary length
      let length := rectangle (0.0, inf);
    end;
  end;
end;
```

B. Some Definitions and Results from Fuzzy Set Theory and Petri Nets

In this chapter we present some definitions and results from fuzzy set theory and Petri nets, as they are used throughout the thesis. Omitted proofs can be found in the literature.

B.1. Fuzzy Set Theory

Fuzzy set theory originates in the seminal paper of Zadeh (1965). This presentation is based on a set of books (Bandemer and Gottwald, 1993; Biewer, 1997; Yager and Filev, 1994; Fedrizzi and Kacprzyk, 1999). In the following, the closed and open interval between a and b is notated as $[a, b]$ and as $]a, b[$, respectively.

B.1.1. Fuzzy sets

A conventional set A may be equated with its *characteristic function*

$$\varphi_A : \mathcal{U} \rightarrow \{0, 1\}$$

associating each element u of a universe of discourse $\mathcal{U}$ a number $\varphi_A(u) \in \{0, 1\}$, such that $\varphi_A(u) = 0$ means that $u \in \mathcal{U}$ does not belong to A, and that $\varphi_A(u) = 1$ means that u belongs to A.

Definition B.1 (Fuzzy Set) *A* fuzzy set *generalizes the binary-valued characteristic function of a conventional set by allowing partial memberships, expressed by a* fuzzy set membership function*:*

fuzzy set

fuzzy set membership function

$$\mu_A : \mathcal{U} \rightarrow [0, 1] \tag{B.1}$$

such that $\mu_A(u) \in [0, 1]$ *is the degree of membership of an element* $u \in \mathcal{U}$ *to the fuzzy set A ranging from* $\mu_A(u) = 0$*, meaning that u does not belong to A, to* $\mu_A(u) = 1$*, meaning that u belongs fully to A, with possible intermediate* $(0 < \mu_A(u) < 1)$ *values.*

It is equivalent to say that a *fuzzy set* A in a universe of discourse $\mathcal{U}$ is defined as a set of pairs

$$A = \{(u, \mu_A(u)); u \in \mathcal{U}\} \tag{B.2}$$

with the membership function $\mu_A : \mathcal{U} \to [0,1]$ of A and $\mu_A(u) \in [0,1]$ defining the grade of membership of $u \in \mathcal{U}$ in A.

Remark B.1 Usually the membership functions use the real interval of $[0,1]$ as co-domain, but this is not necessary. Goguen (1969) showed that a lattice is sufficient as co-domain, so generally fuzzy set A is characterized by its membership function

$$\mu_A : \mathcal{U} \to L.$$

where L is a partially ordered set. Trivially, the unit interval $[0,1]$ is such a partially ordered set.

fuzzy singleton

Notation B.2 If the universe $\mathcal{U}$ of fuzzy set A is finite, i.e., $\mathcal{U} = \{u_1, u_2, \ldots, u_n\}$, the pair $(u, \mu_A(u))$ is denoted by $\mu_A(u)/u$ and called a *fuzzy singleton*. The fuzzy set A is then written as

$$\begin{aligned} A &= \{(u, \mu_A(u))\} = \{\mu_A(u)/u\} \\ &= \mu_A(u_1)/u_1 + \mu_A(u_2)/u_2 + \cdots + \mu_A(u_n)/u_n = \sum_{i=1}^{n} \mu_A(u_i)/u_i \end{aligned} \tag{B.3}$$

where $+$ and $\sum$ are meant in the set-theoretic sense. Pairs with $\mu_A(u) = 0$ are usually omitted.

normal

Fuzzy set A defined in $\mathcal{U}$ is said to be *normal*, iff

$$\sup_{u \in \mathcal{U}} \mu_A(u) = 1, \tag{B.4}$$

subnormal

otherwise A is *subnormal*,

Notation B.3 (Fuzzy Power Set) The set of all fuzzy sets in X, the fuzzy power set of X, is notated as $\mathcal{F}(X)$.

Corollary B.4 The empty set $\emptyset$ and the full set X are elements of $\mathcal{F}(X)$.

Definition B.1 makes no further assumptions on membership functions of fuzzy sets. In practice, some general useful normalized convex functions have been established, which are widely used. Let us first define convexity for fuzzy sets.

convex

Definition B.5 *Let A be a fuzzy set with membership function μ_A and a totally ordered universe $\mathcal{U}$. A is called* convex*, if μ_A has only one maximum:*

$$a, b, c \in \mathcal{U} : a \leq c \leq b \Rightarrow \mu_A(c) \geq \min\{\mu_A(a), \mu_A(b)\} \tag{B.5}$$

In fuzzy control often piece-wise linear functions are used, because they are easy to compute. Such functions are trapezoid, triangle and rectangle functions. Zadeh presented parametric functions which approximate the right and the left flanks of a bell-shape and their combinations. Their graphs are shown in figure B.1 on page 282.

Definition B.6 (Linear Functions) *The rectangle, triangle and trapezoidal parametric functions are assembled from various linear functions with domain $\mathbb{R}$, which is also the universe of the respective fuzzy sets.*

$$rect(x, \alpha, \beta) = \begin{cases} 0 & x < \alpha \\ 1 & \alpha \leq x \leq \beta \\ 0 & x > \beta \end{cases} \tag{B.6}$$

$$triangle(x, m, \alpha, \beta) = \begin{cases} 0 & (x \leq m - \alpha) \vee (x \geq m + \beta) \\ \dfrac{x - (m - \alpha)}{\alpha} & (m - \alpha) < x \leq m \\ 1 - \dfrac{x - m}{\beta} & m < x < (m + \beta) \end{cases} \tag{B.7}$$

$$trapezoid(x, m_1, m_2, \alpha, \beta) = \begin{cases} 0 & (x \leq m_1 - \alpha) \vee (x \geq m_2 + \beta) \\ \dfrac{x - (m_1 - \alpha)}{\alpha} & (m_1 - \alpha) < x \leq m_1 \\ 1 & m_1 \leq x \leq m_2 \\ 1 - \dfrac{x - m_2}{\beta} & m_2 < x < (m_2 + \beta) \end{cases} \tag{B.8}$$

Definition B.7 (Bell-Shape Functions) *Zadeh defined the parametric s-, z-, sz- and π-functions (Biewer, 1997, p. 57/58)) with $x \in \mathbb{R}$, which is also the universe of the respective fuzzy sets.*

$$s(x, \alpha, \gamma) = \begin{cases} 0 & x \leq \alpha \\ 2(\dfrac{x - \alpha}{\gamma - \alpha})^2 & \alpha < x \leq \dfrac{\alpha + \gamma}{2} \\ 1 - 2(\dfrac{x - \gamma}{\gamma - \alpha})^2 & \dfrac{\alpha + \gamma}{2} < x \leq \gamma \\ 1 & otherwise \end{cases} \tag{B.9}$$

$$z(x, \alpha, \gamma) = 1 - s(x, \alpha, \gamma) \tag{B.10}$$

$$sz(x, \alpha, \beta, \gamma, \delta) = \begin{cases} s(x, \alpha, \beta) & x \leq \beta \\ z(x, \gamma, \delta) & x > \beta \end{cases} \tag{B.11}$$

$$\pi(x, \beta, \gamma) = sz(x, \gamma - \beta, \gamma, \gamma, \gamma + \beta) \tag{B.12}$$

Definition B.8 (Support, Core, Height) *The* support *of fuzzy set A is defined as*

support

$$\text{supp } A = \{u \in \mathcal{U} \mid \mu_A(u) > 0\}, \tag{B.13}$$

the core *is defined as*

core

$$\text{core } A = \{u \in \mathcal{U} \mid \mu_A(u) = 1\}, \tag{B.14}$$

the height *is defined as*

height

$$\text{height } A = sup_{u \in \mathcal{U}} \, \mu_A(u). \tag{B.15}$$

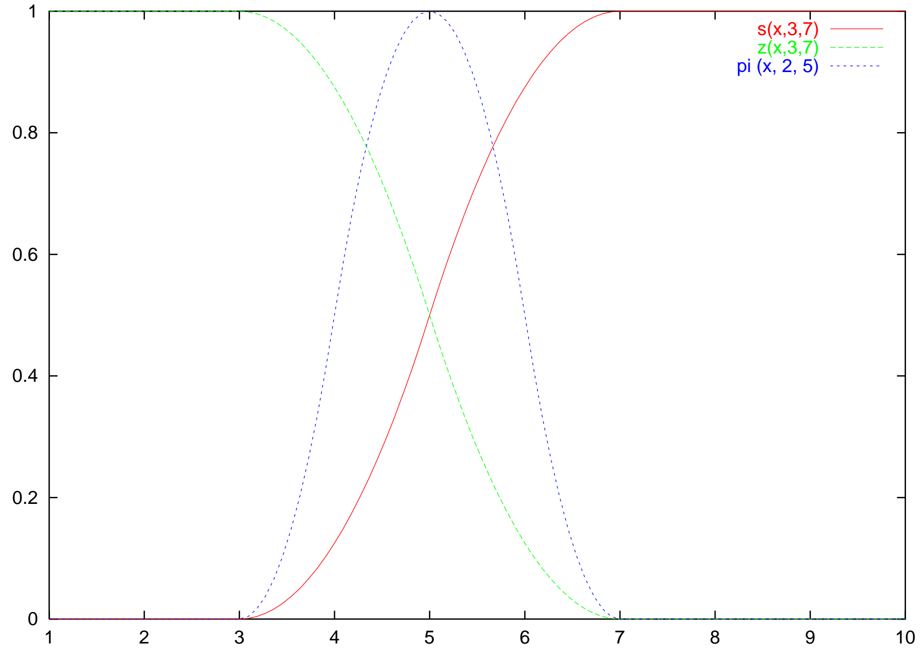

Figure B.1.: The s-, z- and π-functions

Support and core of a fuzzy set are special crisp (i.e. non-fuzzy) subsets defined by the grade of membership. Their generalization is called an alpha-cut.

alpha-cut

Definition B.9 (Alpha-Cut) *The* alpha-cut *of fuzzy set A is denoted as A_α and is defined as*

$$A_\alpha = \{u \in \mathcal{U} \mid \mu_A(u) \geq \alpha\} \qquad \alpha \in]0,1] \tag{B.16}$$

The non-fuzzy subset relation between fuzzy sets A and B holds, if for each element a in the support of A the membership grade of a in B is higher and equal or higher, resp.

subset relation

Definition B.10 (Subset) *Let A, B be fuzzy sets. The non-fuzzy* subset relation $A \subset B$ *holds iff*

$$A \subset B :\Leftrightarrow \forall a \in \operatorname{supp} A : \mu_A(a) < \mu_B(a). \tag{B.17}$$

The subset-or-equal relation is defined in a similar way:

$$A \subseteq B :\Leftrightarrow \forall a \in \operatorname{supp} A : \mu_A(a) \leq \mu_B(a). \tag{B.18}$$

The usual set-theoretic operations of union, intersection and complement are also available for fuzzy sets. They result in new fuzzy sets and are defined by their membership functions.

Definition B.11 (Set theoretic operations) *Let A, B be fuzzy sets. The* union $C = A \cup B$ *is defined with the membership function*

union

$$\mu_C(x) \mathrel{\widehat{=}} \max(\mu_A(x), \mu_B(x)), \tag{B.19}$$

the intersection $C = A \cap B$ *is defined as*

intersection

$$\mu_C(x) \mathrel{\widehat{=}} \min(\mu_A(x), \mu_B(x)) \tag{B.20}$$

and the complement $C = \bar{A}$ *is defined as*

complement

$$\mu_C(x) \mathrel{\widehat{=}} 1 - \mu_A(x). \tag{B.21}$$

The definitions above using max, min for union and intersection are widely used, but other definitions are possible and used. In particular the t-norms and s-norms for intersection and union, resp., are often used.

Definition B.12 (t-norm) *A* t-norm *is a function defined as*

t-norm

$$t : [0,1] \times [0,1] \rightarrow [0,1] \tag{B.22}$$

such that for each $a, b, c \in [0,1]$*:*

1. 1 *is the unit element:* $t(a, 1) = a$.
2. t *is monotone:* $a \leq b \Longrightarrow t(a, c) \leq t(b, c)$.
3. t *is commutative or symmetric:* $t(a, b) = t(b, a)$.
4. t *is associative:* $t(t(a, b), c) = t(a, t(b, c))$.

Definition B.13 (s-norm) *A* s-norm *is a function defined as*

s-norm

$$s : [0,1] \times [0,1] \rightarrow [0,1] \tag{B.23}$$

such that for each $a, b, c \in [0,1]$*:*

1. 0 *is the unit element:* $s(a, 0) = a$.
2. s *is monotone:* $a \leq b \Longrightarrow s(a, c) \leq s(b, c)$.
3. s *is commutative or symmetric:* $s(a, b) = s(b, a)$.
4. s *is associative:* $s(s(a, b), c) = s(a, s(b, c))$.

Relevant examples for t- and s-norms are:

t-norm	$t(a, b)$	*s-norm*	$s(a, b)$
minimum	$\min(a, b)$	maximum	$\max(a, b)$
algebraic product	$a \cdot b$	probabilistic product	$a + b - ab$
Łukasiewicz	$\max(0, a + b - 1)$	Łukasiewicz	$\min(a + b, 1)$

B.1.2. Fuzzy Logic

A direct link exists between fuzzy set theory and fuzzy logic, i.e. a logic of fuzzy truth-values. The expression $x \in A$ is a predicate concerning the membership of x in A. If A is a fuzzy set, then the truth value τ is given by the evaluation of the membership function, i.e.

$$\tau(x \in A) = \mu_A(x).$$

More general, suppose "u is P" is a predicate where P is an imprecise term characterized by a fuzzy set in $\mathcal{U}$, i.e. $P \in \mathcal{F}(\mathcal{U})$. Let $\tau($"u is P"$) = \tau(u, P) = \mu_P(u)$ for each $u \in \mathcal{U}$. In the following we notate with a slight abuse of notation $\tau(P) := \tau(\cdot, P)$. The following definitions are usually employed:

Definition B.14 *Let P, Q be predicates as defined above, t and s are fixed t- and s-norms, respectively. The* negation *of P ("not P"), denoted by $\neg P$, is defined as*

$$\tau(\neg P) = 1 - \tau(P). \tag{B.24}$$

The intersection *of P and Q ("P and Q"), denoted by $P \wedge Q$, is defined as*

$$\tau(P \wedge Q) = t(\tau(P), \tau(Q)), \tag{B.25}$$

where t is a t-norm (B.22)*. The* union *of P and Q ("P or Q"), denoted by $P \vee Q$, is defined as*

$$\tau(P \vee Q) = s(\tau(P), \tau(Q)), \tag{B.26}$$

where s is a s-norm (B.23)*.*

B.1.3. Fuzzy Relations

A fuzzy relations R between non-fuzzy sets X and Y is in turn a fuzzy set of the Cartesian product of X and Y.

fuzzy relation

Definition B.15 *A* fuzzy relation *R between non-fuzzy sets $X_1, \ldots, X_n$ is defined by*

$$R = \{((x_1, \ldots, x_n), \mu_R(x_1, \ldots, x_n)); \quad x_i \in X_i \text{ for } 1 \le i \le n\} \tag{B.27}$$

where $\mu_R : X_1 \times \cdots \times X_n \to [0, 1]$ is the membership function of R.

As a fuzzy relation is a fuzzy set, all definitions, properties etc. on fuzzy sets presented above hold as well.

The composition of two fuzzy relations R in $X \times Y$ and S in $Y \times Z$ is given by the max-min-composition. Certainly the max- and min-operations can be replaced by corresponding t- and s-norms, resp., resulting in a t-s-composition.

Definition B.16 (Max-Min-Composition) *The* max-min-composition *of two fuzzy relations R in $X \times Y$ and S in $Y \times Z$, denoted as $R \circ S$ is defined as fuzzy relation in $X \times Z$ such that*

max-min-composition

$$\mu_{R \circ S}(x, z) = \max_{y \in Y}(\min(\mu_R(x, y), \mu_S(y, z))) \quad \text{for each } x \in X, z \in Z \tag{B.28}$$

Definition B.17 *The* Cartesian product *of two fuzzy sets A in X and B in Y, denoted as $A \times B$, is defined as the fuzzy set in $X \times Y$ such that*

Cartesian product

$$\mu_{A \times B} = \min(\mu_A(x), \mu_B(y)) \quad \text{for each } x \in X, y \in Y \tag{B.29}$$

Definition B.18 *Let $Y = X_1 \times \cdots \times X_{i-1} \times X_{i+1} \times \cdots \times X_n$ and $y = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ for some fuzzy sets X_i. We define the* projection *R_i of fuzzy relation R on $X_1, \ldots, X_n$ as*

projection

$$\mu_{R_i}(x) = \sup\{\mu_R(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n) \mid x_j \in X_j, 1 \leq j \leq n, j \neq i\} \tag{B.30}$$

Definition B.19 *Fuzzy relation R on $X_1, X_2, \ldots, X_n$ is said to* decomposable *iff it can represented as*

decomposable

$$\mu_R(x_1, x_2, \ldots, x_n) = \min\{\mu_{R_1}, \ldots, \mu_{R_i}, \ldots, \mu_{R_n}\} \quad \text{where } x_i \in X_i, 1 \leq i \leq n. \tag{B.31}$$

B.1.4. The Extension Principle

If there exists some relationship between non-fuzzy entities, it is interesting to see, if there exists an equivalent relationship between fuzzy entities. Zadeh's *extension principle* deals with this question.

Definition B.20 (Extension Principle) *Let $A_1, \ldots, A_n$ be fuzzy sets in $X_1, \ldots, X_n$, resp., and*

$$f : X_1 \times \cdots \times X_n \rightarrow Y$$

be some non-fuzzy function such that $y = f(x_1, \ldots, x_n)$. The extension principle *induces the fuzzy set B in Y by $A_1, \ldots, A_n$ via extending the non-fuzzy function f to the fuzzy function f^* such that $B = f^*(A_1, \ldots, A_n)$:*

extension principle

$$f^* : \mathcal{F}(X_1) \times \cdots \times \mathcal{F}(X_n) \rightarrow \mathcal{F}(Y) \tag{B.32}$$

$$\mu_B(y) = \sup_{\substack{(x_1, \ldots, x_n) \in X_1 \times \cdots \times X_n \\ y = f(x_1, \ldots, x_n)}} \min\{\mu_{A_1}(x_1), \ldots, \mu_{A_n}(x_n)\} \tag{B.33}$$

together with $\sup(\emptyset) = 0$.

The extension principle shares the property of other fuzzy set operations, such as union or intersection, that if the A_i are crisp sets (i.e. $A_i = 1.0/x_i$), then B is also crisp with $B = 1.0/y$, and $y = f^*(A_i) = f(x_1, \ldots, x_n)$.

B.1.5. Fuzzy Numbers and Intervals

Fuzzy numbers and intervals are generalizations of ordinary (real) numbers. The corresponding universe is therefore the real line $\mathbb{R}$. Important for all fuzzy numbers and intervals is that their membership function μ is convex.

fuzzy number

fuzzy interval

Definition B.21 *Fuzzy set $A \in \mathcal{F}(\mathbb{R})$ is a* fuzzy number *if A is convex and normalized with exactly one $a \in \mathbb{R}$ such that $\mu_A(a) = 1$. Is A only convex and normalized, then A is a* fuzzy interval.

The basic arithmetical operations with fuzzy numbers are an application of the extension principle (def. B.20 on the page before). The resulting fuzzy sets are also fuzzy numbers. The same operations can also be applied to fuzzy intervals which results also in fuzzy intervals.

Definition B.22 (Basic arithmetical operations) *Let A and B be fuzzy numbers. The* sum *$S := A \oplus B$ is defined with membership function*

$$\forall a \in \mathbb{R}: \quad \mu_S(a) = \sup_{x\in\mathbb{R}} \min\{\mu_A(x), \mu_B(a - x)\}, \tag{B.34}$$

the difference *$D := A \ominus B$ is analogous*

$$\forall a \in \mathbb{R}: \quad \mu_D(a) = \sup_{x\in\mathbb{R}} \min\{\mu_A(x), \mu_B(x - a)\}, \tag{B.35}$$

and also the product *$P := A \odot B$*

$$\forall a \in \mathbb{R}: \quad \mu_P(a) = \sup_{\substack{x,y\in\mathbb{R}\\ xy=a}} \min\{\mu_A(x), \mu_B(y)\}. \tag{B.36}$$

If $0 \notin \operatorname{supp}(B)$, then the quotient *$Q := A \oslash B$ is defined with membership function*

$$\forall a \in \mathbb{R}: \quad \mu_Q(a) = \sup_{\substack{x,y\in\mathbb{R}\\ x/y=a}} \min\{\mu_A(x), \mu_B(y)\}. \tag{B.37}$$

Minimum and maximum of A and B are defined as

$$\forall z \in \mathbb{R}: \quad \mu_{\min}(z) = \sup_{\substack{x,y\in\mathbb{R}\\ \min(x,y)=z}} \min\{\mu_A(x), \mu_B(y)\} \tag{B.38}$$

$$\forall z \in \mathbb{R}: \quad \mu_{\max}(z) = \sup_{\substack{x,y\in\mathbb{R}\\ \max(x,y)=z}} \min\{\mu_A(x), \mu_B(y)\} \tag{B.39}$$

Corollary B.23 Let A be a fuzzy number. The negative $N := \bar{}A$ of A is given by

$$\forall a \in \mathbb{R}: \quad \mu_N(a) = \mu_A(-a). \tag{B.40}$$

The scalar product of a fuzzy number is similar to scalar products of vectors.

Definition B.24 *Let A be a fuzzy number and $\lambda \in \mathbb{R} \setminus \{0\}$. The* scalar product λA *of λ with A is defined with the membership function $\mu_{\lambda A}$*

$$\forall a \in \mathbb{R} : \quad \mu_{\lambda A}(a) = \mu_A(\frac{a}{\lambda}) \tag{B.41}$$

For efficient calculation often piece-wise linear membership functions are used. Details of this approach are given by Dubois and Prade (1987), and Biewer (1997).

For given fuzzy number or interval X four fuzzy semi-intervals exists describing the sets of all numbers lower or greater than X (Dubois and Prade, 1989). We have four intervals, because we allow also the equality case.

Definition B.25 *Let X be a fuzzy number or interval with membership function μ_X. The fuzzy set $[X, +\infty[$ of numbers which are greater or equal than X is defined by the membership function $\mu_{[X,+\infty[}$:*

$$\forall x \in \mathbb{R} : \quad \mu_{[X,+\infty[}(x) = \sup_{y \leq x} \mu_X(y), \tag{B.42}$$

the fuzzy set $]X, +\infty[$ of number which are strictly greater than X is defined by the membership function $\mu_{]X,+\infty[}$:

$$\forall x \in \mathbb{R} : \quad \mu_{]X,+\infty[}(x) = \inf_{y \geq x}(1 - \mu_X(y)), \tag{B.43}$$

The fuzzy set $]-\infty, X]$ of numbers which are lower or equal than X is defined by the membership function $\mu_{]+\infty,X]}$:

$$\forall x \in \mathbb{R} : \quad \mu_{]+\infty,X]}(x) = \sup_{y \geq x} \mu_X(y) = 1 - \mu_{]X,+\infty[}(x), \tag{B.44}$$

the fuzzy set $]-\infty, X[$ of number which are strictly lower than X is defined by the membership function $\mu_{]-\infty,X[}$:

$$\forall x \in \mathbb{R} : \quad \mu_{]-\infty,X[}(x) = \inf_{y \leq x}(1 - \mu_X(y)) = 1 - \mu_{[X,+\infty[}(x). \tag{B.45}$$

B.1.6. Possibility Theory

This section is based on (Dubois and Prade, 1999; Biewer, 1997). Possibility theory uses fuzzy set theory not as a measure for impreciseness but as a measure for uncertainty, i.e. it is a theory of incomplete information similar to probability theory.

Let $\mathcal{U}$ be a universe and P be a (fuzzy) predicate on $\mathcal{U}$, i.e. P is a fuzzy set on $\mathcal{U}$, $P : \mathcal{U} \to [0, 1]$. The proposition "$y$ is P" for $y \in \mathcal{U}$ evaluates to $\mu_P(y)$. Let $x \in \mathcal{U}$ be fixed with unknown exact value but with possible values given by P. The possibility

possibility distribution

that "$u = x$" for u ranging through $\mathcal{U}$ is given by the *possibility distribution* π_x, which is induced by the predicate P, our only source of knowledge about x. The possibility distribution π_x is defined via μ_P:

$$\pi_x(u) = \mu_P(u) \quad \forall u \in \mathcal{U} \tag{B.46}$$

with the following interpretation:

1. $\pi_x(u) = 0$: u can not be assigned to x, short: $x = u$ is impossible.
2. $\pi_x(u) = 1$: u can perfectly be assigned to x, short: $x = u$ is (completely) possible.
3. $\pi_x(u_1) > \pi_x(u_2)$: prefer $x = u_1$ to $x = u_2$ because $x = u_1$ is with a higher degree possible than $x = u_2$.

In contrast to probability distributions, the possibility distributions do not sum up to 1, usually they are normalized resulting in a sum greater than 1. An event with a low probability can have a high degree of possibility and a high degree of possibility induces not automatically a high probability. The degree of possibility is always equal or greater as the probability of the same event.

The measures of possibility and necessity as defined as follwos

possibility

Definition B.26 (Possibility and Necessity) *Let $A \in \mathcal{F}(\mathcal{U})$ and π_x be a (reference) possibility distribution on $\mathcal{U}$, associated with variable x. The* possibility *of "x is A", notated as $\Pi(A|x)$, is defined as*

$$\Pi(A|x) = \sup_{u \in \mathcal{U}} \min\{\mu_A(u), \pi_x(u)\}. \tag{B.47}$$

It measures the degree of compatibility (or consistency) between an element of A and the possibility distribution π_x. The necessity *$\mathcal{N}$ of A is defined as*

necessity

$$\mathcal{N}(A|x) = 1 - \Pi(\neg A|x) = 1 - \sup_{u \in \mathcal{U}} \min\{1 - \mu_A(u), \pi_x(u)\}. \tag{B.48}$$

similarity

For the *similarity* of fuzzy set D (for data) and of reference possibility distribution P (for pattern) over some universe $\mathcal{U}$ we use the definition of Orchard (1999):

$$simil(D, P) = \begin{cases} \Pi(P|D) & \text{if } \mathcal{N}(P|D) > 0.5 \\ (\mathcal{N}(P|D) + 0.5) \cdot \Pi(P|D) & \text{otherwise} \end{cases} \tag{B.49}$$

B.2. Petri Nets

In this section we present some important definitions and properties of Petri nets. Firstly, we present multi-sets followed by Petri nets.

B.2.1. Multi-Sets

For colored nets we need individual tokens at the same place p instead of anonymous tokens as in place-transition nets. These individual tokens are modeled with multi-sets.

Definition B.27 (Multi-sets) *A* multi-set*, M, over base type A is a mapping $A \to \mathbb{N}$. The set of all multi-sets over base type A is denoted with $\mathcal{M}(A)$. Let M, M_1, M_2 be multi-sets over base type A, $a \in A$. Some set-theoretic operations and notations can be transfered to multi-sets; subscript ms indicates that we operate on multi-sets:*

multi-set

1. $M, M_1, M_2 \in \mathcal{M}(A)$
2. $a \in_{ms} M :\Leftrightarrow M(a) > 0$
3. $M_1 \cup_{ms} M_2 = \{(a, M_1(a) + M_2(a)) \mid a \in A\}$
4. $|M|_{ms} = \sum_{a \in M} M(a)$
5. $M_1 \subseteq_{ms} M_2 :\Leftrightarrow \forall a \in A : M_1(a) \leq M_2(a)$
6. $M_1 \setminus_{ms} M_2 = \{(a, M_1(a) - M_2(a) \mid a \in A\}$ *iff* $M_2 \subseteq M_1$
7. $\beta \cdot M = \{(a, \beta n) \mid (a, n) \in M\}$ *for* $\beta \in \mathbb{N}$

We may omit subscript *ms*, if it is clear from the context that we have multi-sets.

The effect of firing of transitions is to remove tokens from multi-set M_1 and to add new token to multi-set M_2. We denote this as formal sums.

Definition B.28 (Formal Sum) *Let M_1 and M_2 be multi-sets over some base type A, $a \in A$. The* formal sums *$M_1 + M_2$ and $M_1 - M_2$ are defined componentwise:*

formal sums

$$\begin{aligned} M_1 + M_2 &= \{(a, M_1(a) + M_2(a)) \mid a \in A\} \\ M_1 - M_2 &= \{(a, M_1(a) - M_2(a)) \mid a \in A\} \text{ iff } M_2 \subseteq M_1 \end{aligned}$$

Let $M = \{M_1, M_2, \dots, M_n\}$ be a set of multi-sets over base type A, we extend the binary formal sum to an n-ary sum:

$$\sum_{m \in M} m = M_1 + M_2 + \cdots + M_n$$

Definition B.29 *Let $A_1 = \{a_1, \dots, a_k\} \subseteq A$. As shorthand notation for multi-set constructors we have*

$$\{(a_1, n_1), \dots, (a_k, n_k)\} = M_1$$

defined as

$$M_1(a) = \begin{cases} n_i & \text{if } a \in A_1 \wedge (a, n_i) \in M_1 \\ 0 & \text{otherwise} \end{cases}.$$

If all $n_i = 1$, we may omit n_i for the sake of brevity: $\{a_1, \ldots, a_k\}_{ms}$. Singleton multi-sets can be written as $(a, n)_{ms}$.

Sometimes, we use the notation by Jensen (1997) for elements of a multi-set:

$$n`a \Leftrightarrow (a, n) \in M.$$

This notation is easier to read if a is a complex data structure.

B.2.2. Petri Nets

B.2.2.1. Basic Petri Nets

According to newer presentations (Rozenberg and Engelfriet, 1998; Desel and Reisig, 1998; Baumgarten, 1996) we distinguish between Petri nets and Petri systems, the former describes the static structure only and the latter defines the dynamic semantics of a net by means of markings and firing behavior.

Petri net
places
transitions
arcs
flow relation

Definition B.30 (Petri Net) *A* Petri net *is a tuple (P, T, A) where P and T are finite sets, $P \cap T = \emptyset$, and $A \subseteq (P \times T) \cup (T \times P)$. Elements of P and T are named* places *and* transitions*, resp., A is the set of* arcs *or the* flow relation*.*

Graphically we denote places, transitions and arcs by circles, rectangles and lines with an arrowhead, respectively.

In the following, we define some important operations on the net structure.

pre-set
post-set
neighborhood

Definition B.31 *Let (P, T, A) be a Petri net. Let $X = P \cup T$ and $x \in X$. The* pre-set *of x, $\bullet x$, is defined as $\bullet x = \{y \in X \mid y\,A\,x\}$. The* post-set *of x, $x\bullet$, is defined as $x\bullet = \{y \in X \mid x\,A\,y\}$. The* neighborhood *of x is defined as $nb(x) = \bullet x \cup x\bullet$.*

We now extend the definitions of pre- and post-sets to their transitive closure, being the basis of properties acyclic and connected nets.

Definition B.32 (Net Structure 1) *Let $N = (P, T, A)$ be a Petri net, $x \in X = P \cup T$, $\bullet^1 x = \bullet x$ and*

$$\bullet^n x = \{y \mid y \in X \wedge y \in \bullet x' \wedge x' \in \bullet^{n-1} x\} \quad n > 1,$$

transitive
closure

$x\bullet^n$ is defined accordingly. The transitive closure *of $\bullet x$ is defined as*

$$\bullet^+ x = \bigcup_{n \geq 1} \bullet^n x,$$

the transitive closure of $x\bullet$ as

$$x\bullet^+ = \bigcup_{n \geq 1} x\bullet^n.$$

acyclic

The net is acyclic*, if*

$$\forall x \in X : x \notin (\bullet^+ x \cup x\bullet^+).$$

Definition B.33 (Net Structure 2) *Let $N = (P, T, A)$ and $N' = (P', T', A')$ be Petri nets with $P = P'$, $T = T'$ and $A' = A \cup A^{-1}$, i.e., N' is the undirected version of N. Net N is* connected *iff*

connected

$$\forall x, x' \in X = P \cup T : x \neq x' \Rightarrow (x, x') \in A'^{+}$$

In the net structure, we can identify root and leaf places in the usual graph theoretical sense.

Definition B.34 *Let $N = (P, T, A)$ be a Petri net. The* root places*, R, of N are defined as*

root places

$$\mathit{root}(N) = \{p \in P \mid \bullet p = \emptyset\},$$

the leaf places*, L, of N are defined as*

leaf places

$$\mathit{leaf}(N) = \{p \in P \mid p\bullet = \emptyset\}.$$

The *marking*, M, of a Petri net (P, T, A) describes the state of a Petri system by assigning values, also called *tokens*, to places. Details of the marking depends on the Petri system type: for elementary nets the marking a map with range is $\{0, 1\}$: $M : P \to \{0, 1\}$, for place-transition nets (P/T-nets) it is a map with range $\mathbb{N}$: $M : P \to \mathbb{N}$. For higher level nets such as Colored Petri Nets (Jensen, 1997) the marking of a place is a multi-set of a datatype's instances. The *initial marking* is notated with M_0.

marking

tokens

initial marking

The *firing behavior* of a Petri net defines the dynamic semantics by stating how we proceed from one to another marking. It defines when a transition is *enabled* which is a precondition of the *occurence* of the transition. If a transition t occurs, it fires, removing tokens from $\bullet t$ and adding new tokens to $t\bullet$. Adding and removing tokens to and from a place need to be properly defined for each net type.

firing behavior

enabled

occurence

Notation B.35 We denote a firing step from marking M_i to M_{i+1}, where transition t fires by, $M_i[t\rangle M_{i+1}$. For a sequence, s, of transitions $s = t_1 \dots t_n$, we denote with $M_i[s\rangle M_j$ that marking M_{i+1} is reached by $M_i[t_1\rangle M_{i+1}$ and recursively the following holds: $M_{i+1}[t_2 \dots t_n\rangle M_j$. The set of all markings reachable from a particular marking M is denoted by $[M\rangle$.

B.2.2.2. Extensions

There are a few well known extensions of the aforementioned basic Petri nets, which are often used. These extension modify the firing behavior of P/T-nets and can be extended to higher-level nets such as Colored Petri nets.

While not explicitely defined above, in P/T-system a firing transition t removes only one token of each place in $\bullet t$ and add exactly one token to each place in $t\bullet$. To generalize the amount of tokens removed and added during firing of transition we introduce arc weights:

arc weights

Definition B.36 (Arc weights) *A Petri net with* arc weights *assigns to each arc a natural number, denoting how many tokens are removed or added from place of the arc if the transition of the arc fires.*

arc expressions

In higher level nets, token usually have a more complex structure than natural numbers as in P/T-nets. Thus, arc weights are extended to *arc expressions* denoting not only the amount of tokens but also an expression defining the values of tokens consumed or produced. Here, a inscription language is needed to formulate the expression, classical example are ML for Colored Petri Nets Jensen (1997) or predicate logic for Pr/T-Nets Smith (1998).

Arc weights define a constant amount of token removed or created. Sometimes it is desirable to have a dynamic arc weight, and in particular an arc weight that is equal to the current number of tokens in the place of interest. This is modeled by reset arcs, which are arcs from places to transitions modifying the firing behavior such that all tokens from the place are removed, the place's marking is reset to an empty marking. To distinguish formally reset arcs from regular arcs, we introduce a new element of the Petri net structure, the set of reset arcs.

reset arcs

Definition B.37 (Net with Reset Arcs) *A net with* reset arcs *is a quadruple* (P, T, A, R) *such that* (P, T, A) *is a Petri net and* $R \subseteq P \times T$.

We denote reset arcs graphically by a double arrow head at the transition end.

Sometimes, we have situations where a transition t shall only fire if a specific place p has an empty marking. This can be expressed by an inhibitor arc from p to t. In contrast to ordinary arcs between transitions and places, no token movement occurs on an inhibitor. To distinguish formally inhibitor arcs from regular arcs, we introduce a new element of the Petri net structure, the set of inhibitor arcs.

inhibitor arcs

Definition B.38 (Net with Inhibitor Arcs) *A Petri net with* inhibitor arcs *is a quadruple* (P, T, A, I) *such that* (P, T, A) *is a Petri net and* $I \subseteq P \times T$.

We denote inhibitor arcs graphically by a circle instead of an arrow at the transition end.

Bibliography

S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors. *Handbook of Logic in Computer Science*, volume 4. Clarendon Press, Oxford, 1995.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1987.

Yahya Y. Al-Salqan and Carl K. Chang. Temporal relations and synchronization agents. *IEEE Multimedia*, pages 30–39, 1996.

Klaus Alfert, Ernst-Erich Doberkat, and Corina Kopka. Towards constructing a flexible multimedia environment for teaching the history of art. SWT-Memo 101, Lehrstuhl für Software-Technologie, Fachbereich Informatik, Universität Dortmund, September 1999. Available online `http://ls10-www.cs.uni-dortmund.de/`.

Klaus Alfert and Matthias Heiduck. Natural language-based specification and fuzzy logic for the multimedia development process. In Wolfgang Gaul and Günter Ritter, editors, *Classification, Automation, and New Media. Proceedings of the 24th Annual Conference of the Gesellschaft für Klassifikation e.V., University of Passau, March 15-17, 2000*, Studies in Classification, Data Analysis, and Knowledge Organization, Berlin, 2002. Springer-Verlag.

James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

James F. Allen and Patrick J. Hayes. A common-sense theory of time. In *9th International Joint Conference on Artificial Intelligence*, pages 528–531, 1985.

J. C. M. Baeten and C. Verhoef. Concrete process algebra. In Abramsky et al. (1995), pages 149–268.

Brian P. Bailey, Joseph A. Konstan, and John V. Carlis. DEMAIS: Designing multimedia applications with interactive storyboards. In *ACM Multimedia 01. The 9th ACM International Multimedia Conference*, pages 241–250, Ottawa, Onatario, Canada, September/October 2001a.

Brian P. Bailey, Joseph A. Konstan, and John V. Carlis. Supporting multimedia designers: Towards more effective design tools. In Lloyd Ruthledge, editor, *Proceeding of the 8th International Conference on Multimedia Modeling 2001 (MMM01)*, Amsterdam, The Netherlands, November 2001b.

Hans Bandemer and Siegfried Gottwald. *Einführung in Fuzzy-Methoden.* Akademie Verlag, Berlin, 4th edition, 1993.

Bernd Baumgarten. *Petri-Netze: Grundlagen und Anwendungen.* Hochschultaschenbuch. Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 2nd edition, 1996.

Christoph Begall. VOLT – Ein Übersetzer für Vitruv_L. Diplomarbeit (Master's thesis), Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie, 2002. In preparation, In German.

Benno Biewer. *Fuzzy-Methoden. Praxisrelevante Rechenmodelle und Fuzzy-Programmiersprachen.* Springer-Verlag, Berlin, 1997.

Berry W. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61–72, May 1988.

Susanne Boll and Wolfgang Klas. $\text{Z}_\text{Y}\text{X}$ – a multimedia document model for reuse and adaption of multimedia content. *IEEE Transactions of Knowledge and Data Engineering*, 13(10):361–382, May 2001.

Grady Booch, James Rumbaugh, and Ivar Jacobsen. *The unified modeling language user guide.* Object Technology Series. Addison-Wesley, 1999.

Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C Recommendation REC-xml-19980210, World Wide Web Consortium, February 1998.

Manfred Broy. Formalization of distributed, concurrent, reactive systems. In Neuhold and Paul (1991), pages 319–362.

Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllighofen. *Prototyping: An Approach to Evolutionary System Development.* Springer-Verlag, Berlin edition, 1992.

Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, Boca Raton, 1997.

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

Janette Cardoso and Heloisa Camargo, editors. *Fuzziness in Petri Nets*, volume 22 of *Studies in Fuzziness and Soft Computing*. Physica-Verlag, Heidelberg, New York, 1999.

Janette Cardoso, Robert Valette, and Didier Dubois. Fuzzy petri nets: An overview. In *13th IFAC World Congress, San Francisco, USA*, volume J, pages 443–448, June/July 1996.

Edmund M. Clarke, Jr, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambrigde, MA, 1999.

Donald D. Cowand and Carlos J. P. Lucena. Abstract data views: An interface specification concept to enhance design for reuse. *IEEE Transactions on Software Engineering*, 21(3):229–243, March 1995.

Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

Jörg Desel and Wolfgang Reisig. Place/transition petri nets. In Reisig and Rozenberg (1998), pages 122–173.

Michel Diaz and Patrick Sénac. Time stream petri nets – a model for timed multimedia information. In Robert Valette, editor, *Application and Theory of Petri Nets 1994. 15th International Conference*, number 815 in LNCS, pages 219–238. Springer-Verlag, 1994.

Ernst-Erich Doberkat. A language for specifying hyperdocuments. *Software – Concepts and Tools*, 17:163–172, 1996.

Ernst-Erich Doberkat and Dietmar Fox. *Software Prototyping mit SETL*. Leitfäden und Monographien der Informatik. BG Teubner Verlag, Stuttgart, 1989.

Ernst-Erich Doberkat, Wolfgang Franke, Wilhelm Hasselbring, Claus Pahl, Hand-Gerald Sobottka, and Bettina Sucrow. PROSET – Prototyping with Sets. Language Definition. Technical report, Lehrstuhl für Software-Technologie, Fachbereich Informatik, Universität Dortmund, 1994.

Didier Dubois and Henri Prade. Fuzzy numbers: An overview. In J. C. Bezdek, editor, *Analysis of Fuzzy Information Vol. 1: Mathematics and Logic*, pages 3–39. CRC Press, Boca Raton (FL), 1987.

Didier Dubois and Henri Prade. Processing fuzzy temporal knowledge. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(4):729–744, July/August 1989.

Didier Dubois and Henri Prade. A brief introduction to possibility theory and its use for processing fuzzy temporal information. In Cardoso and Camargo (1999), pages 52–71.

Didier Dubois, Henri Prade, and Florence Sèdes. Fuzzy logic techniques in multimedia database querying: A prelimary investigation of the potentials. *IEEE Transactions of Knowledge and Data Engineering*, 13(10):383–392, May 2001.

Didier Dubois, Henri Prade, and Ronald R. Yager, editors. *Readings in Fuzzy Sets for Intelligent Systems.* Morgan Kaufmann, San Mateo (CA), 1993.

Soumitra Dutta. An event based fuzzy temporal logic. In *Proceedings 18th International Symposium on Multiple-Valued Logic*, pages 64–71, Palma de Mallorca, Spain, 1988. IEEE Computer Society.

E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier Science Publishers B. V., Amsterdam, New York, Oxford, Tokyo, 1990.

Bruce A. Epstein. *Director in a Nutshell.* O'Reilly, 1999.

A. Fantechi, S. Gnesi, G. Ristori, M. Cerenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. *Formal Methods of System Design*, 4(3):243–263, 1994.

Alexander Fay and Eckehard Schnieder. Fuzzy petri nets for knowledge modeling in expert systems. In Cardoso and Camargo (1999), pages 300–318.

Mario Fedrizzi and Janusz Kacprzyk. A brief introduction to fuzzy sets and fuzzy systems. In Cardoso and Camargo (1999), pages 25–51.

A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.

William Fleming and David John Watkin. The history of western architecture: The renaissance. In *Britannica CD '99.* Encyclopædia Britannica, Inc., 1999.

Martin Fowler and Kandal Scott. *UML distilled: A brief guide to the standard object modeling language.* Addison-Wesley, Reading, MA, 2nd edition, 1999.

Franca Garzotto, Paolo Paolini, and Daniel Schwabe. HDM — a model-based approach to hypertext application design. *ACM Transactions on Information Systems*, 11(1):1–26, January 1993.

Vincenzo Gervasi. *Environment Support for Requirements Writing and Analysis.* PhD thesis, Dipartimento di Informatica, Università degli Studi di Pisa, 2000.

Vincenzo Gervasi and Bashar Nuseibeh. Lightweight validation of natural language requirements. *Software — Practice and Experience*, 32(2):113–133, February 2002.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, NJ, 1991.

Simon J. Gibbs and Dionysios C. Tsichritzis. *Multimedia Programming – Objects, Environments and Frameworks.* Addison-Wesley, 1995.

Rainer Goetze. *Dialogmodellierung für multimediale Benutzerschnittstellen.* Teubner-Texte zur Informatik. Teubner, Stuttgart [u.a.], 1. korr. Nachdr. edition, 1995. ISBN 3-8154-2064-4. Dissertation, Universität Oldenburg.

Joseph Goguen. The logic of inexact concepts. *Synthese*, 19:325–373, 1969. reprinted in Dubois et al. (1993).

James Gosling and Ken Arnold. *Java: The Language.* Addison-Wesley, 1996.

Christopher Habel, Michael Herweg, and Simone Pribbenow. Wissen über Raum und Zeit. In Günther Görz, editor, *Einführung in die künstliche Intelligenz*, chapter 1.4, pages 139–204. Addison-Wesley, Bonn, 1993.

Elżbieta Hajnicz. *Time Structures. Formal Description and Algorithmic Representation.* Number 1047 in Lecture Notes in Artificial Intelligence, subseries of LNCS. Springer-Verlag, Berlin, 1996.

Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *Communications of the ACM*, 37(2):30–39, February 1994.

Komei Harada, Eiichiro Tanaka, Ryuichi Ogawa, and Yoshinori Hara. Anecdote: A multimedia storyboarding system with seamless authoring support. In *ACM Multimedia 96. The 4th ACM International Multimedia Conference*, pages 341–351, Boston, MA, USA, 1996.

Lynda Hardman, Dick C. A. Bulterman, and Guido van Rossum. The amsterdam hypermedia model: Adding time and context to the dexter modell. *Communications of the ACM*, 37(2):50–62, February 1994.

David Harel. Statechards: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

Matthias Heiduck. Konzeption einer Beschreibungssprache für eine eingeschränkte Klasse von Multimedia-Systemen. Diplomarbeit (Master's thesis), Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie, July 1999. In German, Available online `http://ls10-www.cs.uni-dortmund.de/`.

C. A. R. Hoare. *Communicating sequential processes.* Prentice Hall International Series in Computer Science. Prentice Hall, 1985.

Karen Holtzblatt and H. Beyer. Making customer-centered design work for teams. *Communications of the ACM*, 36(10), October 1993.

Ellis Horowitz. *Fundamentals of Programming Languages.* Computer Science Press, 2nd edition, 1984.

Philipp Hoschka. Synchronized multimedia integration language (SMIL) 1.0 specification. W3C Recommendation REC-smil-19980615, World Wide Web Consortium, June 1998.

Anthony Hunter and Bashar Nuseibeh. Managing inconsistent specifications: Reasoning, analysis and action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, October 1998.

Tomǎs Isakowitz, Edward A. Stohr, and P. Balasubramanian. RMM: A methodology for structured hypermedia design. *Communications of the ACM*, 38(8):34–44, August 1995.

Jens H. Jahnke and Conor Ryan, editors. *Proceedings of the 2nd International Workshop on Soft Computing Applied to Software Engineering (SCASE)*, Enschede, The Netherlands, February 2001. University of Twente, ISBN 90-365-1552-1.

Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 2nd edition, 1997. 2nd corr. reprint.

Peter Jonsson and Christer Bäckström. A unifying approach to temporal constraint reasoning. *Artificial Intelligence*, 102:143–155, 1998.

Sarah Kahn and Michael J. Muller. Participatory design – introduction to the special section. *Communications of the ACM*, 36(6):24–28, June 1993.

H. Khalfallah and A. Karmouch. An architecture and a data model for integrated multimedia documents and presentation applications. *ACM Multimedia Systems*, 3 (5/6):238–250, 1995.

John F. Koegel Buford. *Multimedia Systems.* ACM-Press, Reading, MA, 1994a.

John F. Koegel Buford. Uses of multimedia information. In Koegel Buford (1994a), pages 1–25.

Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques.* John Wiley and Sons, 1998.

Olaf Kummer, Frank Wienberg, and Michael Duvigneau. Renew – user guide. Release 1.5.2, Distributed Systems Group, Theoretical Foundations Groups, Department for Informatics, University of Hamburg, July 2001. Available online at `http://www.renew.de`.

Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

T. D. C. Little and A. Ghafoor. Inverval-based conceptual models for time-dependent multimedia data. *IEEE Transactions of Knowledge and Data Engineering*, 5(4):551–563, August 1993.

Thomas D. Little and Arif Ghafoor. Synchronzation and storage models for multimedia objects. *IEEE Journal on Selected Areas in Communication*, 8(3):413–427, April 1990.

Thomas D. C. Little. Time-based media representation and delivery. In Koegel Buford (1994a), chapter 7, pages 175–200.

Xiaoqing Frank Liu and John Yen. An analytic framework for specifying and analyzing imprecise requirements. In *Proceedings of ICSE 18*, pages 60–69, 1996.

Carl G. Looney. Fuzzy petri nets fo rule-based decisionmaking. *IEEE Transactions on Systems, Man, and Cybernetics*, 18(1):178–183, January 1988. reprinted in Cardoso and Camargo (1999).

David Lowe and Wendy Hall. *Hypermedia & the Web: An Engineering Approach*. John Wiley and Sons, 1999.

Kim Halskov Madsen and Peter H. Aiken. Experiences using cooperative interactive storyboard prototyping. *Communications of the ACM*, 36(6):57–64, June 1993.

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. ACM-Press. Addison-Wesley, Reading, MA, 1993.

Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

John C. Mitchel. *Foundations for Programming Languages*. Foundations of Computing. MIT Press, Cambrigde, MA, 2nd. printing 1998 edition, 1996.

Stephen J. Morris and Anthony C. W. Finkelstein. Integrating design and development in the production of multimedia documents. In Mülhäuser and Effelsberg (1996), pages 98–108.

Stephen J. Morris and Anthony C. W. Finkelstein. Engineering via discourse: Content structure as an essential component for multimedia documents. *International Journal of Software Engineering and Knowledge Engineering*, 9(6):691–724, 1999.

Peter D. Mosses. A practical introduction to denotational semantics. In Neuhold and Paul (1991), pages 1–50.

Max Mülhäuser and Wolfgang Effelsberg, editors. *MMSD '96: Proceedings of the International Workshop on Multimedia Software Development, March 26/26 1996, Berlin*, Los Alamitos, CA, March 1996. IEEE Press.

Tadao Murata. Temporal uncertainty and fuzzy-timing high-level petri nets. In Jonathan Billington and Wolfgang Reisig, editors, *Application and Theory of Petri Nets 1996. 17th International Conference*, number 1091 in LNCS, pages 11–28. Springer-Verlag, 1996.

Tadao Murata, Takeshi Suzuki, and Sol M. Shatz. Fuzzy-timing high-level petri nets (fthns) for time-critical systems. In Cardoso and Camargo (1999), pages 88–114.

Francis Neelamkavil. *Computer Simulation and Modeling*. John Wiley and Sons, 1987.

Erich J. Neuhold and Manfred Paul, editors. *Formal Description of Programming Concepts*. IFIP State-Of-The-Art Reports. Springer-Verlag, Berlin, 1991.

Steven R. Newcomb, Neill A. Kipp, and Victoria T. Newcomb. The hytime hypermedia/time-based document structuring language. *Communications of the ACM*, 34(11):67–83, November 1991.

Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

OMG. *OMG Unified Modeling Language Specification*. Object Management Group, Inc., September 2001. Version 1.4. Available online at `http://www.omg.org`.

Robert A. Orchard. *NRC Fuzzy Java Toolkit User's Guide*. Integrated Reasoning, Institute for Information Technology, National Research Council Canada, 1.0 beta 1 edition, November 1999. available online at `http://ai.iit.nrc.ca/IR_public/fuzzy/`.

Helmuth Partsch. *Requirements-Engineering systematisch. Modellbildung für softwaregestützte Systeme*. Springer-Verlag, Berlin, 1998.

Fabiano Borges Paulo, Paulo Cesar Masiero, and Maria Cristina Ferreira de Oliveira. Hypercharts: Extended statecharts to support hymermedia specification. *IEEE Transactions on Software Engineering*, 25(1):33–49, January/February 1999.

Dimitrie O. Păun and Marsha Chechik. Events in linear-time properties. In William N. Robinson and Kevin Ryan, editors, *4th IEEE International Symposium on Requirements Engineering*, pages 123–132, Limerick, Ireland, June 1999.

Amir Pnueli. The temporal logics of programs. In *Proc. 18th Ann. IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

Klaus Pohl. Requirements engineering: An overview. Aachener Informatik-Berichte 96-5, RWTH Aachen, Fachgruppe Informatik, 1996.

Chris Reade. *Elements of functional programming.* International Computer Sciences Series. Addison-Wesley, 1989.

Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1998.

C. Rolland and C. Proix. A Natural Language Approach For Requirements Engineering. In P. Loucopoulos, editor, *Proceedings of the Fourth International Conference CAiSE'92 on Advanced Information Systems Engineering*, volume 593, pages 257–277, Manchester, United Kingdom, 1992. Springer-Verlag. URL `citeseer.nj.nec.com/rolland92natural.html`.

Grzegorz Rozenberg and Joost Engelfriet. Elementary net systems. In Reisig and Rozenberg (1998), pages 12–121.

James Rumbaugh, Grady Booch, and Ivar Jacobsen. *The unified modeling language reference manual.* Object Technology Series. Addison-Wesley, 1999.

C. A. S. Santos, P. N. M. Sampaio, and J. P. Courtiat. Revisiting the concept of hypermedia document consistency. In *ACM Multimedia 99 (Part 2). The 7th ACM International Multimedia Conference*, pages 183–186, Orlando, FL, USA, November 1999.

Stefan Sauer and Gregor Engels. Extending UML for modeling of multimedia applications. In *Proceedings of the 1999 IEEE International Symposium on Visual Languages (VL'99)*, pages 80–87, Tokyo, Japan, September 1999.

David A. Schmidt. *The Strucutre of Typed Programming Languages.* Foundations of Computing. MIT Press, Cambrigde, MA, 1994.

Daniel Schwabe and Gustavo Rossi. The object-oriented hypermedia design model. *Communications of the ACM*, 38(8):45–46, August 1995.

Daniel Schwabe, Gustavo Rossi, and Simone D. J. Barbosa. Abstraction, composition and lay-out definition mechanisms in OOHDM. In I. F. Cruz, J. Marks, and K. Wittenburg, editors, *Electronic Proceedings of the ACM Workshop on Effective Abstractions in Multimedia. In Association with ACM Multimedia '95*, San Francisco, California,

November 1995. Electronic address: `http://www.cs.tufts.edu/~isabel/mmwsproc.html`.

Einar Smith. Principles of high-level net theory. In Reisig and Rozenberg (1998), pages 175–210.

Ian Sommerville. *Software-Engineering.* Addison-Wesley, 4th edition, 1990.

J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

Ralf Steinmetz. *Multimedia-Technologie. Grundlagen, Komponenten und Systeme.* Springer-Verlag, Berlin, 3rd edition, October 2000.

Alison Stones. Glossary of medieval art and architecture. Www-document, University of Pittsburgh, 1997. available online at `http://www.pitt.edu/~medart/menuglossary/INDEX.HTM`.

Marc Störzel. Simulation verteilter Prozesslandschaften. Diplomarbeit (Master's thesis), Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie, October 2001.

Marc Störzel and Klaus Alfert. *Benutzerhandbuch "Renew-Auswertungskomponente".* Universität Dortmund, Fachbereich Informatik, Lehrstuhl für Software-Technologie, January 2002. In German, unpublished.

Bjarne Stroustrup. *The C++ programmling language.* Addison-Wesley, 2nd edition, 1991.

Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.

Helmut Thiele. Einführung in die Fuzzy-Logik. Scriptum zur Spezialvorlesung, Universität Dortmund, Fachbereich Informatik, 1998.

Klaus Tochtermann. *Ein Modell für Hypermedia. Beschreibung und integrierte Formalisierung wesentlicher Hypermediakonzepte.* PhD thesis, Universität Dortmund, 1994.

Michalis Vazirgiannis. *Interactive multimedia documents: modeling, authoring, and implementation experiences*, volume 1564 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1999.

Michalis Vazirgiannis, Yannis Theodiridis, and Timos Sellis. Spatio-temporal composition in multimedia applications. In Mülhäuser and Effelsberg (1996), pages 120–127.

Marc Vilain and Henry Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of American Association for Artificial Intelligence*, pages 377–382, 1986.

William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1999.

Glynn Winskel. *The Formal Semantics of Programming Languages. An Introduction*. Foundations of Computing. MIT Press, Cambrigde, MA, 1993.

Glynn Winskel and Mogens Nielson. Models for concurrency. In Abramsky et al. (1995), pages 1–148.

Ronald R. Yager. Fuzzy temporal methods for video multimedia information systems. *Journal of Advanced Computational Intelligence*, 1(1):37–44, 1997.

Ronald R. Yager and Dimitar P. Filev. *Essentials of Fuzzy Modeling and Control*. John Wiley and Sons, 1994.

Lofti A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965. reprinted in Dubois et al. (1993).

Y. Zhou, T. Murata, T. DeFanti, and T. Zhang. Fuzzy-timing petri net modeling and simulation of a networked virtual environemnt - NICE. *IEICE Transactions Fundamentals Special Section on Concurrent Systems Technology*, 83(11):2166–2175, September 2000.

Yi Zhou and Tadao Murata. Fuzzy-timing petri net model for distributed multimedia synchronization. In *Proceedings of the 1998 IEEE International Conference on Systems, Man, and Cybernetics*, pages 244 – 249, Lolla, California, October 1998.

Yi Zhou and Tadao Murata. Petri net model with fuzzy-timing and fuzzy-metric temporal logic. *International Journal of Intelligent Systems*, 14(8):719–746, August 1999.

/99/ T. Bühren, M. Cakir, E. Can, A. Dombrowski, G. Geist, V. Gruhn, M. Gürgrn, S. Handschumacher, M. Heller, C. Lüer, D. Peters, G. Vollmer, U. Wellen, J. von Werne
Endbericht der Projektgruppe eCCo (PG 315)
Electronic Commerce in der Versicherungsbranche
Beispielhafte Unterstützung verteilter Geschäftsprozesse
Februar 1999

/100/ A. Fronk, J. Pleumann,
Der DoDL-Compiler
August 1999

/101/ K. Alfert, E.-E. Doberkat, C. Kopka
Towards Constructing a Flexible Multimedia Environment for Teaching the History of Art
September 1999

/102/ E.-E. Doberkat
An Note on a Categorial Semantics for ER-Models
November 1999

/103/ Christoph Begall, Matthias Dorka, Adil Kassabi, Wilhelm Leibel, Sebastian Linz, Sascha Lüdecke, Andreas Schröder, Jens Schröder, Sebastian Schütte, Thomas Sparenberg, Christian Stücke, Martin Uebing, Klaus Alfert, Alexander Fronk, Ernst-Erich Doberkat
Abschlußbericht der Projektgruppe PG-HEU (326)
Oktober 1999

/104/ Corina Kopka
Ein Vorgehensmodell für die Entwicklung multimedialer Lernsysteme
März 2000

/105/ Stefan Austen, Wahid Bashirazad, Matthais Book, Traugott Dittmann, Bernhard Flechtker, Hassan Ghane, Stefan Göbel, Chris Haase, Christian Leifkes, Martin Mocker, Stefan Puls, Carsten Seidel, Volker Gruhn, Lothar Schöpe, Ursula Wellen
Zwischenbericht der Projektgruppe IPSI
April 2000

/106/ Ernst-Erich Doberkat
Die Hofzwerge — Ein kurzes Tutorium zur objektorientierten Modellierung
September 2000

/107/ Leonid Abelev, Carsten Brockmann, Pedro Calado, Michael Damatow, Michael Heinrichs, Oliver Kowalke, Daniel Link, Holger Lümkemann, Thorsten Niedzwetzki, Martin Otten, Michael Rittinghaus, Gerrit Rothmaier
Volker Gruhn, Ursula Wellen
Zwischenbericht der Projektgruppe Palermo
November 2000

/108/ Stefan Austen, Wahid Bashirazad, Matthais Book, Traugott Dittmann, Bernhard Flechtker, Hassan Ghane, Stefan Göbel, Chris Haase, Christian Leifkes, Martin Mocker, Stefan Puls, Carsten Seidel, Volker Gruhn, Lothar Schöpe, Ursula Wellen
Endbericht der Projektgruppe IPSI
Februar 2001

/109/ Leonid Abelev, Carsten Brockmann, Pedro Calado, Michael Damatow, Michael Heinrichs, Oliver Kowalke, Daniel Link, Holger Lümkemann, Thorsten Niedzwetzki, Martin Otten, Michael Rittinghaus, Gerrit Rothmaier
Volker Gruhn, Ursula Wellen
Zwischenbericht der Projektgruppe Palermo
Februar 2001

/110/ Eugenio G. Omodeo, Ernst-Erich Doberkat
Algebraic semantics of ER-models from the standpoint of map calculus.
Part I: Static view
März 2001

/111/ Ernst-Erich Doberkat
An Architecture for a System of Mobile Agents
März 2001

/112/ Corina Kopka, Ursula Wellen
Development of a Software Production Process Model for Multimedia CAL Systems by Applying Process Landscaping
April 2001

/113/ Ernst-Erich Doberkat
The Converse of a Probabilistic Relation
Oktober 2002

/114/ Ernst-Erich Doberkat, Eugenio G. Omodeo
Algebraic semantics of ER-models in the context of the calculus of relations.
Part II: Dynamic view
Juli 2001

/115/ Volker Gruhn, Lothar Schöpe (Eds.)
Unterstützung von verteilten Softwareentwicklungsprozessen durch integrierte Planungs-, Workflow- und Groupware-Ansätze
September 2001

/116/ Ernst-Erich Doberkat
The Demonic Product of Probabilistic Relations
September 2001

/117/ Klaus Alfert, Alexander Fronk, Frank Engelen
Experiences in 3-Dimensional Visualization of Java Class Relations
September 2001

/118/ Ernst-Erich Doberkat
The Hierarchical Refinement of Probabilistic Relations
November 2001

/119/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer, Ingo Röpling,
Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Intermediate Report
November 2001

/120/ Volker Gruhn, Ursula Wellen
Autonomies in a Software Process Landscape
Januar 2002

/121/ Ernst-Erich Doberkat, Gregor Engels (Hrsg.)
Ergebnisbericht des Jahres 2001
des Projektes "MuSofT – Multimedia in der SoftwareTechnik"
Februrar 2002

/122/ Ernst-Erich Doberkat, Gregor Engels, Jan Hendrik Hausmann, Mark Lohmann, Christof Veltmann
Anforderungen an eine eLearning-Plattform — Innovation und Integration —
April 2002

/123/ Ernst-Erich Doberkat
Pipes and Filters: Modelling a Software Architecture Through Relations
Juni 2002

/124/ Volker Gruhn, Lothar Schöpe
Integration von Legacy-Systemen mit Eletronic Commerce Anwendungen
Juni 2002

/125/ Ernst-Erich Doberkat
A Remark on A. Edalat's Paper *Semi-Pullbacks and Bisimulations in Categories of Markov-Processes*
Juli 2002

/126/ Alexander Fronk
Towards the algebraic analysis of hyperlink structures
August 2002

/127/ Markus Alvermann, Martin Ernst, Tamara Flatt, Urs Helmig, Thorsten Langer
Ingo Röpling, Clemens Schäfer, Nikolai Schreier, Olga Shtern
Ursula Wellen, Dirk Peters, Volker Gruhn
Project Group Chairware Final Report
August 2002

/128/ Timo Albert, Zahir Amiri, Dino Hasanbegovic, Narcisse Kemogne Kamdem,
Christian Kotthoff, Dennis Müller, Matthias Niggemeier, Andre Pavlenko, Stefan Pinschke,
Alireza Salemi, Bastian Schlich, Alexander Schmitz
Volker Gruhn, Lothar Schöpe, Ursula Wellen
Zwischenbericht der Projektgruppe Com42Bill (PG 411)
September 2002

/129/ Alexander Fronk
An Approach to Algebraic Semantics of Object-Oriented Languages
Oktober 2002

/130/ Ernst-Erich Doberkat
Semi-Pullbacks and Bisimulations in Categories of Stochastic Relations
November 2002

/131 Yalda Ariana, Oliver Effner, Marcel Gleis, Martin Krzysiak,
Jens Lauert, Thomas Louis, Carsten Röttgers, Kai Schwaighofer,
Martin Testrot, Uwe Ulrich, Xingguang Yuan
Prof. Dr. Volker Gruhn, Sami Beydeda
Endbericht der PG nightshift:
Dokumentation der verteilten Geschäftsprozesse im FBI und Umsetzung von Teilen dieser Prozesse im Rahmen eines FBI-Intranets basierend auf WAP- und Java-Technologie
Februar 2003

/132/ Ernst-Erich Doberkat, Eugenio G. Omodeo
ER Modelling from First Relational Principles
Februar 2003

/133/ Klaus Alfert, Ernst-Erich Doberkat, Gregor Engels (Hrsg.)
Ergebnisbericht des Jahres 2002 des Projektes "MuSofT – Multimedia in der SoftwareTechnik"
März 2003

/134/ Ernst-Erich Doberkat
Tracing Relations Probabilistically
März 2003

/135/ Timo Albert, Zahir Amiri, Dino Hasanbegovic, Narcisse Kemogne Kamdem,
Christian Kotthoff, Dennis Müller, Matthias Niggemeier,
Andre Pavlenko, Alireza Salemi,Bastian Schlich, Alexander Schmitz,
Volker Gruhn, Lothar Schöpe, Ursula Wellen
Endbericht der Projektgruppe Com42Bill (PG 411)
März 2003

/136/ Klaus Alfert
Vitruv: Specifying Temporal Aspects of Multimedia Presentations —
A Transformational Approach based on Intervals
April 2003

/137/ Klaus Alfert, Jörg Pleumann, Jens Schröder
A Framework for Lightweight Object-Oriented Design Tools
April 2003