Conceptual Content Modeling

Languages, Applications, and Systems

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
genehmigte Dissertation

von
Sebastian Boßung
aus Marburg

2008

Gutachter:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Heinrich C. Mayr

Tag der mündlichen Prüfung: 22. Februar 2008

Summary

Contemporary content management systems enable authors to express their intentions and ideas not only textually but by means of a variety of multimedial artifacts. This enables dualistic descriptions of real-world entities as requested by the German philosopher Ernst Cassirer (1874-1945). Cassirer's *symbols* jointly describe entities by a conceptual model and a multimedial representation of the entity. Such dualistic descriptions have been successfully integrated with conceptual models and are used as a foundation of content management systems.

The model presented in this thesis captures semantics of multimedial content in order to provide dualistic descriptions of entities. It makes heavy use of content: as the subject of explanation as well as to express semantics. Entities are also described conceptually whereby individual pieces of content are related and rich descriptions of entities can be created. In contrast to existing approaches, which are mostly aimed at computers, this work presents a user-centric approach to providing dualistic descriptions of real-world entities.

The objectives discussed above are approached by a concept-oriented content model which concentrates on making ASSociations between its terms Explicit and Transitive (ASSET). Asset Expression language and systems provide three contributions: (1) The language to create *Asset Expressions* uses a few simple-to-understand paradigms and is easily transformed into a visual representation for the user. (2) Enhancements to Asset Expressions facilitate the creation of semantic descriptions and the collaboration between users. (3) Processes are available to work with these descriptions, e.g., to guide the mapping of description networks onto software systems that are based on conceptual schemata.

While Asset Expressions syntactically borrow from the λ-calculus, it is not their main purpose to be reducible like λ-expressions. Instead, Asset Expressions denote real-world entities by decomposing their descriptions. The result of this denotation is the idea of the denoted entity in the mind of the viewer. Therefore, it is usually counter-productive to only present a reduced form to the user. While the textual syntax of Asset Expressions is similar to that of the λ-calculus, any medial content is considered a literal and shown directly in the expression. Asset Expressions can easily be transformed into a visual representation that is more appealing to humans.

Abstractions over medial content point out aspects of the entity that need further explanation, for example a person shown in a picture. The purpose of an abstraction is to create structure in the description by making explicit some of the entity's elements. *Applications* connect abstractions to further Asset Expressions and thus provide the explanations requested by the abstractions, for example a rich model giving details about the person. Content can be sub-structured into *content components* over which abstractions can be made to signify the aspect of the entity the abstraction refers to.

Semantic types are used to type Asset Expressions. Types are assigned to abstraction variables and to content. Semantic types have no structural effect on the expression they type: They do not prescribe a certain set of abstractions. The purpose of semantic types is to introduce some semantic restrictions on explanations to reduce the danger of defining meaningless expressions. To facilitate the collaboration of users, templates (called *traits*) for

sets of abstractions along with a semantic type for the expression can be defined. Traits guide the creation of new expressions leading to similar structures in related expressions.

Large numbers of expressions can be created to model an application domain. To retrieve expressions from a larger set or to select parts of existing expressions to create new ones, an *Asset Expression Query Language* is available.

Asset Expressions provide their users with the means to flexibly model entities from an application domain. *Asset Expression Systems* offer a generic platform to handle the lifecycle of Asset Expressions. In these systems, application domains are modeled extensionally, i.e., by the body of expressions in the domain. To create a software system for a domain that is modeled with Asset Expressions, a conceptual schema can be extracted from the expressions by a *user-centric schema construction process*, which is run by the domain experts who provided the expressions. The process automatically creates large parts of the schema and prompts the domain experts with particular questions without the involvement of a modeling expert. Feedback is given in the form of prototype software systems, which allow the assessment of the practicability of the schema.

Contents

List of Figures vii

List of Tables x

1 Introduction 1
- 1.1 Representation of Entities . . . 3
- 1.2 Entity Descriptions in Information Systems . . . 6
- 1.3 Research Objectives . . . 8
  - 1.3.1 Capturing the Meaning of Content . . . 8
  - 1.3.2 System Support . . . 9
- 1.4 Approach to Solution and Overview . . . 10

2 Context of This Work 13
- 2.1 Data Model Extensions . . . 13
  - 2.1.1 Persistent Data . . . 13
  - 2.1.2 Persistency in Programming Languages . . . 15
  - 2.1.3 Assets: Dualistic Description of Entities . . . 17
  - 2.1.4 Conceptual Content Management . . . 18
- 2.2 Foundations . . . 21
  - 2.2.1 Functional Programming . . . 21
  - 2.2.2 Denotational Semantics . . . 23
  - 2.2.3 Ontologies . . . 24
- 2.3 Semantic Models for Multimedial Content . . . 27
  - 2.3.1 Structured Documents . . . 28
  - 2.3.2 Hypermedia . . . 32
  - 2.3.3 Semantic Descriptions . . . 34
- 2.4 Introduction to Some Technologies . . . 38
  - 2.4.1 XML Schema . . . 38
  - 2.4.2 XML Inclusions . . . 39
  - 2.4.3 XML Query . . . 40

3 Core Asset Expression Language 43
- 3.1 Plain Asset Expressions . . . 44
  - 3.1.1 Syntax . . . 45
  - 3.1.2 Visual Notation . . . 48
  - 3.1.3 Lifecycle . . . 49
- 3.2 Content Components . . . 49
  - 3.2.1 Components . . . 51
  - 3.2.2 Selectors in Different Kinds of Content . . . 53
- 3.3 Typed Asset Expressions . . . 59
  - 3.3.1 Semantic Types . . . 60

3.3.2 Type Construction . . . 61
3.3.3 Expression Typing . . . 61
3.3.4 Typing Rules . . . 63
3.3.5 Rationale of This Type System . . . 65
3.3.6 Typing Example . . . 66

4 Extended Asset Expression Language 67
4.1 Traits for Asset Expressions . . . 67
4.2 Asset Expression Query Language . . . 69
4.2.1 Navigating Expressions . . . 70
4.2.2 Predicates . . . 72
4.2.3 Creation . . . 73
4.3 Handling Components . . . 74
4.3.1 Mention-Handling . . . 74
4.3.2 Use-Handling . . . 75
4.3.3 Piece-Handling . . . 75
4.4 Normal Form with Content . . . 75
4.4.1 Reduction . . . 75
4.4.2 Substitution . . . 76
4.5 Content Construction . . . 78
4.6 Notes on the Use of Asset Expressions . . . 78
4.6.1 Multiple Domains . . . 80
4.6.2 Openness and Dynamics . . . 81
4.7 Grammar . . . 81

5 Systems for Asset Expression Support 85
5.1 Scope of Asset Expression Systems . . . 85
5.2 A General Architecture for Asset Expression Systems . . . 86
5.3 Storage Layer . . . 87
5.3.1 Semi-structured Data Model for Asset Expressions . . . 89
5.4 Manipulation Layer . . . 89
5.4.1 Workspaces . . . 89
5.4.2 Content Kind Registry . . . 91
5.4.3 Persistency Points . . . 92
5.4.4 Reduction . . . 93
5.4.5 Query . . . 93
5.5 Presentation Layer . . . 95
5.5.1 Navigation . . . 98
5.5.2 Automatic Layout . . . 99

6 Asset Expressions and Conceptual Schemata 101
6.1 Intensional Typing of Asset Expressions . . . 102
6.1.1 Types . . . 104
6.1.2 Typing Rules . . . 104
6.1.3 Types for Characteristics and Content . . . 106
6.1.4 Typing Example . . . 108
6.2 Converting Asset Expressions to Asset Instances . . . 108
6.2.1 Translation Rules . . . 109
6.2.2 Conversion of Content Components . . . 110
6.2.3 Convertible Expressions . . . 111

7 Pragmatics **113**
7.1 Type Inference for Semantic Types . . . 113
7.1.1 Representation of Signatures . . . 114
7.1.2 Reasoning for Signature Matching . . . 116
7.1.3 Example . . . 121
7.2 Type Inference for Intensional Types . . . 125
7.2.1 Lessons Learnt . . . 125
7.2.2 Process Support for Openness and Dynamics . . . 127
7.2.3 System Integration . . . 135
7.2.4 Quality of the Created Schema . . . 136

8 Application Example **139**
8.1 The Application Domain . . . 139
8.2 Asset Expressions for Domain Entities . . . 140
8.3 Semantic Types . . . 141
8.4 Collaborative Creation of Expressions . . . 143
8.5 Conceptual Schema and Information System . . . 145
8.6 Evaluation . . . 147

9 Summary and Discussion **149**
9.1 Contributions . . . 149
9.2 Comparison with Related Approaches . . . 151
9.2.1 Descriptions of Medial Content . . . 151
9.2.2 Schema Creation Processes . . . 155
9.3 Future Work . . . 156

Bibliography **160**

Index **179**

List of Figures

1.1 Symbols as combination of content and concepts . . . 4
1.2 Dualistic description of entities by Assets . . . 6
1.3 Structure of this thesis around Asset Expressions . . . 10

2.1 Sample of relational data and query . . . 14
2.2 DBPL example . . . 16
2.3 Query expressed as a set comprehension . . . 17
2.4 Module setup for personalization . . . 20
2.5 Typing rules of the simply typed λ-calculus . . . 23
2.6 Ontology spectrum . . . 25
2.7 The On-To-Knowledge meta-process . . . 26
2.8 Markup used for typesetting . . . 28
2.9 An annotation and its base . . . 30
2.10 Conceptual annotations . . . 31
2.11 Linking among hypermedia documents in a web-based hypermedia system . . . 33
2.12 A simple RDF example . . . 35
2.13 The Caliph tool for capturing MPEG-7 media description on photographs . . . 36
2.14 MPEG-7 Semantic Description Scheme . . . 37
2.15 XQuery example . . . 40

3.1 Asset Expressions model entities from the real world . . . 45
3.2 Connecting function parameters to body variables . . . 50
3.3 Component examples . . . 50
3.4 Image with two persons in different roles . . . 51
3.5 Content kinds . . . 54
3.6 Abstraction over a component in XML content . . . 56
3.7 Chaining selectors for heterogenous content . . . 59
3.8 Amount of abstraction and specificity of semantic types . . . 60
3.9 Visual notation for typed Asset Expressions . . . 62
3.10 Semantic typing of Asset Expressions . . . 65

4.1 Traits as combination of abstractions and semantic type . . . 68
4.2 Abstract syntax tree representation of an Asset Expression . . . 71
4.3 Sub-expressions included by the different paths . . . 72
4.4 Content construction . . . 76
4.5 A large Asset Expression network . . . 79
4.6 Combination of semantic types from multiple domains . . . 80

5.1 Three-tier architecture for a system based on Asset Expressions . . . 87
5.2 Logical view of the semi-structured data model for the storage layer . . . 88
5.3 Workspace and its user . . . 90

5.4 Workspace scenario of teacher-student collaboration . . . 90
5.5 Content kind registry . . . 91
5.6 Lifecycle of Asset Expressions in the persistency layer . . . 93
5.7 Asset Expression editor . . . 96
5.8 Drag-and-drop between workspaces . . . 97
5.9 Automatic layout of Asset Expressions . . . 99

6.1 Conversion of freely modeled expressions into intensional instances . . . 102
6.2 Intensional typing of Asset Expressions . . . 107
6.3 An Asset model for content components . . . 111

7.1 Knowledge representation system . . . 114
7.2 Hierarchy of semantic types . . . 115
7.3 Example of an Asset Expression signature modeled in description logic . . . 116
7.4 Concept contraction examples . . . 118
7.5 Concept abduction examples . . . 120
7.6 Semantic type hierarchy for inference example . . . 122
7.7 Expressions for semantic type inference example . . . 123
7.8 Semantic type inference . . . 124
7.9 Incompatible personalizations in GKNS . . . 126
7.10 Process for openness and dynamics . . . 129
7.11 Information system generation from Asset Expressions . . . 130
7.12 Expressions with coinciding structure and their allocation to classes . . . 132
7.13 Component configuration for schema evolution . . . 134
7.14 Combination of AES and CCMS . . . 135
7.15 Dimensions of conceptual schema quality . . . 137

8.1 Venia Legeni for Ludwig Heydenreich . . . 141
8.2 Reuse of domain entities . . . 142
8.3 Summary of semantic types from GKNS . . . 143
8.4 The GKNS user interface . . . 145
8.5 Screenshot of the data entry tool from the GKNS project . . . 146
8.6 The architecture of the GKNS system . . . 147

9.1 Design principles of the Asset Expression approach . . . 150
9.2 Richness and formality of modeling paradigms . . . 157

List of Tables

1.1 Examples of syntactic, semantic, and media objects 2
1.2 Panofsky's methodology of three description levels 5
1.3 Classification of information systems based on Panofsky's levels of description . 7

3.1 Examples of selectors 55
3.2 Selector kinds for plain texts 56
3.3 Selector kinds for images 57
3.4 Selector kinds for audio 57
3.5 Spatio-temporal description features in MPEG-7 58
3.6 Examples of expressions and their type 63

4.1 Axes available in AEQL and the corresponding expressions 71
4.2 Paths of AEQL 72
4.3 Content substitution 77

9.1 Comparison of approaches to interrelate content and/or capture its semantics . 152

Chapter 1

Introduction

Contemporary content management systems enable their users to express ideas and intentions not only textually but by means of a variety of multimedial artifacts. These artifacts capture meaning and pass it on to the future viewer of the artifact. The philosopher Ernst Cassirer notes ([Cas00], citation after [Sch06]):

> "What is actually preserved to us of the past are specific historical monuments—'monuments' in words and writing, in image and bronze. This becomes history for us only once we begin to see in these monuments symbols not only in which we recognize specific forms of life, but by virtue of which we are able to restore them for ourselves."

Cassirer points out that past creations of artists (or—in general—authors) are manifest in 'monuments' which need to be brought back to life (i.e., be interpreted in the correct context by the viewer) to become history. Traditionally, such monuments are works such as paintings, statues, or even music. With the widespread availability of technology to digitally create, handle, and manipulate multimedial content, the importance of digital representations of monuments in multimedial content increases.

Many software system today use multimedial content to accomplish their task. News web sites, e.g., present images and video alongside textual descriptions. It is, however, common for these software systems to treat multimedial content as essentially a raw array of bytes, for which system support is limited to storing and retrieving. Tools for creation and manipulation of content—which must be aware of the parts it is composed of—are usually available externally. If meaning is conveyed in the content, this meaning is not made explicit in the software system. Users of the system reinterpret the content every time it is presented. If the system cannot assist in determining its meaning, pitfalls are manifold. A particular user might misinterpret the content because of wrong assumptions about the context it should be perceived in. Other users might not be able to make sense of the content at all, lacking crucial information.

In recent years capturing the semantics of multimedial content has gained increasing interest from Computer Scientists. As computational resources and the knowledge of using them have become ubiquitous to content creators [GB05], large collections of various content have emerged stressing the need for system support that extends beyond mere storage and retrieval.

To provide such support content management systems should capture the semantics of multimedial content together with the content. Application scenarios span from machine-interpretable medial content—which can for example be automatically classified—all the way to the description of semantics for human users to share an understanding of the content that non-expert users might not have. Capabilities of systems supporting such scenarios therefore need to include:

Syntactic objects **Scott Numerals**	*Semantic objects* **Peano Axioms**	*Media Objects* **Number notations**
Defined in λ-expressions: zero := $\lambda f.\lambda x.x$ succ := $\lambda n.\lambda f.\lambda x(f)((n)f)x$	– 0 is a natural number. – Every natural number a has a single successor, denoted by $succ(a)$. – ...	*fonts and notation systems*
$\lambda f.\lambda x.(f)(f)(f)x$	succ(succ(succ(0)))	Roman: III Arabic: 3 Binary: 11

Table 1.1: *Examples of syntactic, semantic, and media objects*

- The means to capture real-world entities in a computer system by providing multimedial representations of the entities and of their semantic descriptions.
- Aiding users—computers or humans—in understanding these entities when presented with the captured descriptions.

In the case of a human user it is assumed that the understanding still has to happen in the mind of the human who cannot be provided with "pre-understood" content. However, the goal is to make the process of understanding as simple as possible, e.g., by providing visually helpful co-occurrences which the user can "mentally compile and decipher" [Dor03].

When representing entities in a computer system, a typical way to deal with complexity is the structural decomposition of the entities and the creation of networks of such entities. Structural similarities discovered by decomposition are described in a conceptualization of the application domain. This approach is for example taken in class-based conceptual models that define classes as an aggregation of attributes.

Other fields—for example that of programming languages—suggest that the endeavor of capturing semantics is not entirely hopeless. Numbers are a part of most programming languages but their semantics are easier to describe than those of the language as a whole. They will thus serve as an example here. There are *syntactic objects*, namely the numerals which are expressed in some language, for example in the λ-calculus as Scott numerals [Sto77]. These are shown on the left in table 1.1. The syntactic objects have no intrinsic meaning, but there should be (and is in the case of numerals) a mapping to *semantic objects* which is so easy to understand that it seems trivial to the audience. The semantic objects in the example are numbers as defined by the Peano axioms. Finally, *media objects*—such as Arabic or Roman numbers—provide appealing access to human users. Given a certain familiarity with the formalism, humans develop a direct mapping of media and semantic objects. The degree of directness of this mapping highly depends on the appropriateness of the media objects and training of their use.

In the context of system support, the triple of syntactic, semantic, and media objects can be used in the following way: Syntactic objects can be manipulated by the system, they thus need to be expressed in a formalism suitable to the intended manipulations (e.g., arithmetics in the example of numbers). Human users use media objects to interact with the system in which a part of the application domain is represented. This application domain is defined by semantic objects.

Multimedial content is first of all a media object. A system can present such content to human users who make sense of it by interpreting it in the context of their available knowledge. For human users, the backing of media objects with semantic objects will remain in the mind of

the user. The system can, however, be equipped with syntactic objects which allow some manipulations of media objects such that the cognitive load on the user can be reduced. Suitable syntactic objects, their mapping to media objects, and means to facilitate the recognition of the corresponding semantic objects by the user will be developed in this thesis. Several approaches to semantics exist (most prominently axiomatic, denotational, and operational semantics, see, e.g., [Sto77] for an overview). The research and development in content management systems presented in this work combines all three kinds of objects—syntactic, semantic, and media objects—and uses the denotational approach for content modeling.

The term "content" has been used informally above. Its general English meaning is the "significance or meaning of something" [Web90]. Therefore it is clear that "multimedial content" actually refers to a "multimedial content *representation*" as it would make little sense otherwise. This differentiation of *content* and *content representation* is often not made in literature because it is usually apparent from the context which is meant. The terminology below will continue in this fashion, by using the much shorter "content" instead of "multimedial content representation".

It should also be pointed out that the notion of multimedial content adopted here is rather broad. Not only does it include audio and video—which are commonly associated with the term—but any medium that can be used to convey meaning. Examples thus include formatted or plain texts, three-dimensional models, still images, and combinations thereof. Primitive values (e.g., numbers, character strings, dates, ...) are also considered cases of multimedial content. One can then create a coherent modeling paradigm that does not need to differentiate "primitive values" and "medial content".

1.1 Representation of Entities

Information systems[1] are usually built to create a model of some part of the application domain (see, e.g., [BMS84, Bor85]). To this end, the entities of the domain are described abstractly in a model. The model defines the means available within the system to represent real-world entities. Applications of such models are manifold and can be found almost anywhere large amounts of data need to be handled, e.g., in enterprise management systems [EP00], e-commerce applications [DTOP02], or research support systems [SBS05].

Since the introduction of conceptual modeling [Che76, BMS84] the approach of building domain models as a conceptual basis for information systems enjoys widespread popularity.[2] In general, conceptual modeling techniques provide means to abstractly describe the application domain by combining simple structures into larger ones. These structures can usually be related and the process can be repeated to create even more complex models. An important difference of conceptual models and, e.g., type definitions in programming languages is that conceptual models emphasize capturing the application domain without paying attention to implementation details.

The creation of conceptual models is only weakly formalized and resembles—in large parts—more the creation of a work of art than the application of well understood principles [Hal01, 1.1]. To a lesser degree this problem also exists for the later creation of model instances to describe particular elements of the application domain. Both processes are not yet fully understood.

A dualistic occurrence of abstract conceptual model and individual medial representation of a real-world entity seems to be essential for the representation and successful understanding of models of the real world. In fact, Ernst Cassirer notes [SR02] that it is necessary to

[1] "An IS [Information System] is a complex system whose raison d'etre is the set of interactions that it carries out with its environment." [GST00] For these interactions it is usually necessary to model some part of the environment, i.e., the application domain, inside the system. ISs therefore usually contain models of entities found in their environments. Content Management Systems (CMSs) are a particular kind of IS that put special emphasis on the support of multimedial content.

[2] The university of Klagenfurt has compiled a website listing many important steps in the history of conceptual modeling [SMBP].

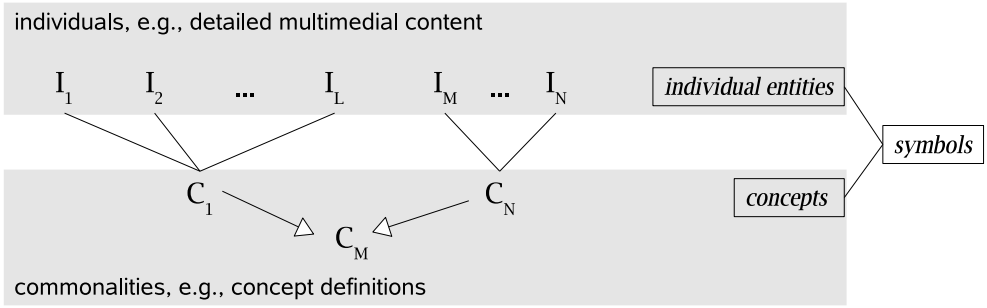


Figure 1.1: *Combination of individual entities which are represented by content and concept to form* symbols.

conjointly describe entities by medial content as well as by conceptual abstraction to appropriately capture an entity from the application domain. Cassirer calls the combination of content and concept a *symbol* (figure 1.1). Symbols by nature need to be defined as pairs of content and concept, either alone is not enough. The concept has no purpose without the backing by actual instances (represented by medial content) as it stands alone as a purely ficticious concept with no relevance in the application domain. Conceptualizations thus require support through (extensional) definitions by medial representations. Medial representations without conceptualization do not provide much value to humans either. They tend to be perceived as an opaque blob of medial features which conveys no meaning because the individual features cannot be cognitively processed for lack of conceptualizations.[3] To further reduce the complexity of descriptions higher-level concepts can be formed. These describe the commonalities of several concepts in an abstract manner.

Examples for the dualism of concept and content can be found in human cognitive processes, which are Cassirer's domain of discourse [Cas01]. Our vision provides us with an image of our surroundings similar to a photograph. This image is composed of various entities. However, we can only recognize these entities if we are familiar with the appropriate conceptualizations. An entity for which no conceptualization exists, because it is unlike anything we have ever seen before, is likely not to be recognized as a discrete entity but just blends with other parts of the image. If the entity reoccurs several times, our mind might be triggered to create a new conceptualization for it that is filled with increasing amounts of detail as we learn about the entity. Therefore, Cassirer does not assume a fixed set of predefined concepts [SR02].

An important insight of Cassirer is that new concepts are formed and existing ones are modified in a continuous process. This process is *open* and *dynamic* in nature [Seh04]. It is open because it allows the extension of the existing set of concepts to suit new description needs and also allows the modification of description instances. It is dynamic because such extensions and modifications can be carried out at any time. Cassirer thereby accounts for the collection of new perceptions and cognitions.

Through their joint description of an entity, medial content and conceptual descriptions do not just stand side-by-side but are interrelated in a symbol. The conceptual abstraction of an entity describes aspects shared by several instances; the medial content resprents the individual entity. Conceptual abstraction and content together form an integrated symbolic entity representation. The dualistic descriptions are related to other descriptions of their kind, forming a network. When capturing meaning in a network of descriptions as requested by Cassirer, medial representation and conceptual abstraction jointly describe entities from the application domain. Therefore, a structurally rich way to capture explanations of content is

[3] An example of this is listening to a conversation in an unfamiliar language.

	Pre-Iconographical Level *generic*	Iconographical Level *specific*	Iconological Level *abstract*
Subject matter	identifiable objects	analysis based on domain knowledge	interpretation based on deep cultural knowledge and understanding
Example	colors, primitive shapes	particular humans	religion

Table 1.2: *Panofsky's methodology of three description levels [Pan70].*

needed to provide system support for such descriptions. Its requirements will be discussed in more detail in section 1.3.

Several researchers have considered which types of information can be conveyed by an image. Many base themselves [CR03] onto the work of Erwin Panofsky [Pan70]. Panofsky distinguishes three levels: *pre-iconographical*, *iconographical*, and *iconological* (table 1.2). The pre-iconographical level describes factual matters that are "apprehended by identifying pure forms".[4] These matters can be understood based on everyday experience. Examples are colors and simple objects which are indifferent to culture. The second level is the iconographical level. This level requires domain knowledge to analyze combinations of factual matter from the pre-iconographic level: "We connect artistic motifs and combinations of artistic motifs (compositions) with themes or concepts." Descriptions on the iconological level are based on cultural knowledge and a deep understanding of the application domain "by ascertaining those underlying principles which reveal the basic attitude of a nation, a period, a class, a religious or philosophical persuasion." Consider an image of the crucifixion of Christ as an example. The image contains pre-iconographical aspects such as wooden bars and a male person, iconographical ones such as Jesus Christ or the crucifixion, and iconological ones such as religion and suffering (example after [Wes04]).

Others have developed schemes that are similar to Panofsky's. Shatford Layne [Lay88] distinguishes *ofness* and *aboutness*. Ofness is analogous to Panofsky's pre-iconographical level as it has to do with objects identifiable in the image itself. Aboutness corresponds to the iconographical level where an analysis based on domain knowledge is performed to find out what the image is about.

To create models of entities, one can, according to Charles Sanders Peirce [Pei31], rely on three dimensions. Peirce calls these *firstness*, *secondness*, and *thirdness* to avoid terms that already carry meaning:

1. *Firstness*: Entities are described in this dimension by properties which are inherent to the entity. These are also called *characteristic* properties. Examples are the physical dimensions of an object, the colors of a painting, etc.

2. *Secondness*: Describes entities in terms of their relationships to external entities. Secondness properties are not intrinsic to an entity as they result from two or more partners which are both entities of their own. The siblings of a person are an example of a relationship between entities.

3. *Thirdness*: Principles or constraints about entities are covered in this dimension. Thirdness is, for example, about the regularity of attributes of entities. Possible future combinations in the secondness dimension are determined by principles described in the thirdness. The fact that all adults are above a certain age is an example of an entity description in the thirdness dimension.

[4] All literal citations in this paragraph from [Pan70, chapter 1].

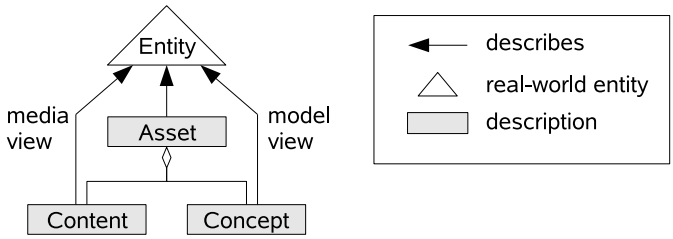


Figure 1.2: *Dualistic description of entities by Assets [Seh04]. An Asset describes a real-world entity and is composed of content and concept. The two parts provide a media and a model view of the entity.*

Dualistic representation of entities as demanded by Cassirer can be combined with the description dimensions of Peirce in a paradigm to conceptually model real-world entities. An example of such dualistic respresentations are *Assets* [Seh04], which combine a medial and a conceptual model view on the entity, see figure 1.2. In accordance with Peirce, the conceptual view is composed of the characteristic properties (firstness), relationships with other entities (secondness), and constraints on the entity which regulate future states of description (thirdness).

1.2 Entity Descriptions in Information Systems

Database management systems (DBMSs) provide well-understood means to efficiently store, manipulate, and retrieve data. Data in these systems can represent domain entities, however, no information is provided about the nature of such a representation. DBMSs excel at composing data of a few primitive types that are combined regularly, for example in the relational data model [Cod70]. On this data, DBMSs provide isolated access in the sense that even though multiple users might be working in the system at the same time, the goal is to create the impression of being alone in the system to each single user. Despite the lack of semantic descriptions of data and the goal of isolation of users, DBMSs are often incorporated into systems that provide user collaboration. Such information systems (IS, [GST00]) add semantics to the data in the database by interpreting them in a consistent way. They also implement collaboration facilities for users by combining means available from the DBMS with other services.

The interpretation of data in a consistent fashion is a central feature of an IS. To this end, ISs are often based on a conceptual model of the application domain which captures the necessary semantics. These semantics then need to be observed throughout the implementation of the system. In most ISs these semantics are assumed to be fixed. In fact, they already exist before the implementation of the system is started, thus forming its basis. This violates Cassirer's assumption that conceptual models should be extendable as one's understanding of the application domain evolves. Systems which account for such an evolution of the conceptual model need to make the modeling facilities explicit to allow their users to modify the model when the need arises. A change in model tears down the foundations of an IS if it has been implemented by traditional means. To survive such a change in a semantically consistent manner, systems need to react to the new model by making it their new foundation.

A system's support for entity representation can be classified according to Panofsky's levels of description as summarized in table 1.3. At the pre-iconographical level, databases support entity description by means of values from common domains. These domains must be shared knowledge between all users as the database does not provide models of the domains. In the relational model, which is the most common database modeling paradigm today, entities are

Level	System	Description means
pre-iconographical	Database management systems	Combinations of common domains, e.g., numbers, dates
iconographical	Information systems with fixed model e.g., digital libraries	Conceptual model which exists a priori and describes application domain as foundation of system
iconological	Information systems with open model e.g., conceptual content management systems	Conceptual model which is open to user modification; appropriate response of the system

Table 1.3: *Classification of information systems based on Panofsky's levels of description*

described as tuples composed of primitive values. In most implementations these values can be from the domains of numbers, character strings, dates, and sometimes also from more specific domains such as geographic points. Tuples of identical structure are aggregated in relations. While relations can be named, this name is rarely sufficient to describe the semantics of the relation. Moreover, the limitations of many implementations—e.g., length restrictions on the name—often enforce meaningless names that allow very little conclusions to be drawn as to the semantics of the relation. Queries can be posed on the pre-iconographical level in terms of primitive values, for example value restrictions on numbers or the colors occurring in pictures if the system is a multimedial database.

Descriptions at the iconographic level require expressiveness in the terms of the application domain. Information systems offer these means to their users, usually based on a conceptual model of the domain. In most contemporary software engineering methodologies this model is hand-crafted into the system by a manual transformation of the model into code. Model-driven methodologies—which are currently the topic of much research [MM03, Sel03]—automate this transformation to a large degree. Nevertheless, the model is inherently coupled with the system. Both approaches to system development require the model to exist before implementation can begin. Queries on the iconographical level are given in the terminology of the model. This often does not only comprise generic query facilities (such as retrieving all tuples with value x from a relational database) but domain-specific queries (such as finding all persons in a certain role) which might not be easily decomposed into queries on primitive values. This decomposition must, however, be ultimately possible as iconographical descriptions are usually broken down into pre-iconographical descriptions. Another example of an iconographical query is the detection of larger entities in content (e.g., athletes in sports videos as in the currently ongoing Boemie project [SPK+05, Boe07]).

Iconological descriptions of entities in an information system are based on deep cultural knowledge. Such interpretations of the application domain are the result of an understanding which influences the description of the domain. An open model as a basis for an information system can support this process as the system needs to adapt to the insight gained in the process. Additionally, it is necessary to incorporate cultural knowledge. As an example, classification taxonomies can capture this knowledge but at the same time can be modified according to the current understanding of the domain. Access to information on the iconological level does not only mean access in terms of the application domain but rather through this domain because the available information is provided in the shape of the current model. As the iconological level includes cultural aspects, it can be expected that groups of users will not always be able to agree on a common model for the description of entities. Providing each user with the opportunity of having a personal, subjective conceptual model of the entitiy—and therefore of the part of the real-world that is modeled in the system—allows users to

collaborate to the extent they agree on a common model, but also to co-exist separately in the remaining differences.

Access to entities on all three of Panofsky's levels is important to answer all types of questions from users. In a study in 1988 [Mar88] it was observed that (then current) information systems largely provide access to pictorial resources on the iconographical or even on the iconological level. This was found to be problematic for people who are not sufficiently familiar with the system's classification or subject heading schemes. If alternative access paths do not exist, such users are essentially locked out of the system. As a remedy it is proposed [Mar88] to put more emphasis on pre-iconographical access paths to the system. However, many users do not want to query for just anything of a certain color or texture, but for larger semantic entities [HTS+06], requiring iconographical or iconological access paths. Integrated access means on all three levels is therefore highly desirable.

1.3 Research Objectives

Capturing real-world entities by medial content is of most benefit to users if done on all three of Panofsky's levels in an integrated fashion. Based on such descriptions, access can be provided to users on all levels. The work presented here focuses on a user-centric approach to providing such descriptions. Very briefly, the objective is to ...

Develop a structurally rich way to capture explanations of content.

This goal will be elaborated below. While the primary audience of the approach consists of human users, computers also need to be addressed because descriptions of multimedial content are best given and handled in a computerized system. On top of this, traditional information systems have benefits of their own—such as distribution, persistency, and access control—which should be made available without reinventing too many wheels.

There exist several approaches that provide semantic descriptions of multimedial content to computers, some of which are discussed in section 2.3. However, human users usually find it difficult to directly work with these semantic descriptions because of mainly two reasons:

1. The structural complexity of the descriptions is commonly high. The description paradigms are intended to capture information and make it available to a machine. Moreover, the paradigms are often chosen because they are easily processable by the machine, resulting in structures that are difficult to understand for human users [SM99, MGMW05].

2. The descriptions given of a multimedial content are too detailed. Many of the details that need to be given explicitly to a machine as a basis for some understanding are obvious to humans [NM06]. Therefore, the details of the resulting large descriptions can hide the actually interesting information.

The remainder of this section will outline some requirements to a human-centric approach to capture meaning of multimedial content and the support of this approach *by* computer system. The latter goal is not the same as the description of multimedial content *to* computers.

1.3.1 Capturing the Meaning of Content

To capture meaning of content—especially if this is to be done in a visually rich manner—one must first be able to capture the multimedial content itself. With vast amounts of work on encodings, multimedia databases, and so forth available, current content management systems excel at storing and retrieving content of various formats. A variety of transformations on content are also available at what is the pre-iconographic level to computers (pixels, audio samples, or strings of text). These capabilities must be available to users, either directly or by interfaces to existing technology.

Next, if a "*structurally rich* way to capture [...] content" is sought, content must be describable not only as an opaque whole. Instead, means to identify and refer to parts of this content are needed. To address parts of content, different addressing schemes depending on the type of content (e.g., two-dimensional images, video, etc.) are available in existing work and can be incorporated. Exploiting the substructure of content in the opposite direction, a mechanism should also be available for the construction of content.

As the content is now readily at hand and even richly structured, one can proceed to capture its meaning. Focussing on human users, a common use case will be to bring descriptions captured by one person (called *prospector*) to the understanding of another (called *inspector*). To this end, the prospector must be able to provide any additional information that is required to understand the content (an *explanation*). The prospector creates expressions which explain the content, possibly also relating them to some particular part of the content—as designated by the rich substructure. For reasons of orthogonality, details of explanations should be supplied using the same mechanism as is used to describe the original content. This will serve to create whole networks of interrelated expressions.

Prospectors then enjoy an extensive freedom in creating descriptive expressions from content. While this enables them to always explain everything in the way most appropriate to content and audience at hand, prospectors have to exercise special care not to produce invalid or meaningless explanations. A similar problem exists in programming languages where users can combine language primitives in any way permissible by the syntax. Programming languages supply type systems which allow users to add some application knowledge which can then be enforced onto the constructed program by a type checker. This prevents a large number of—usually accidental—senseless constructs. Creating similar mechanisms to assert some semantic restrictions for the explanations of content can yield analogous benefits.

1.3.2 System Support

Working with multimedial content is much easier within specialized computer systems. Descriptions, which are created as outlined above, can be supported by such systems in two ways. First of all, a dedicated system allows prospectors to handle, structure, and describe content in order to build the explanatory networks necessary to capture the meaning of the content for inspectors. In addition, large advances have been made over the past decades concerning the support of collaborative work in information systems in general and in content management in particular which should also be made available within this context. Therefore, interfacing content descriptions with existing information systems is highly desirable. These two classes of system support can be broken down into more concrete tasks.

In dedicated systems, the typing of explanations can be utilized not only to increase the odds of building meaningful explanations but can also be used as part of the explanation itself. Similarly to Cassirer's concepts, additional information can be given on the nature of the described entity. Moreover, some work already exists in this area, system support can be given for assigning such *semantic types* to descriptions. The suggestions of types by the system can, for example, be based on the types of other nearby explanations.

In addition to dedicated systems, a transition path to existing information systems is highly desirable. As information systems are usually based on a conceptual model, which does not have to be fixed in all cases, this bridging can happen in two steps: Creating a conceptual model as a basis for the information system and transforming descriptions at the instance level to be imported into the information system. Despite semantic typing, prospectors should be able to create descriptions in the form most appropriate to the task at hand. This means that semantic typing should not have structural consequences for descriptions. Clearly, a transition to a conceptual model is only viable if the descriptions are uniform enough to produce a coherent conceptual model according to which instances can be transformed. It is thus necessary to provide a process which guides the consolidation of the domain and the creation of a conceptual model based on consolidated descriptions.

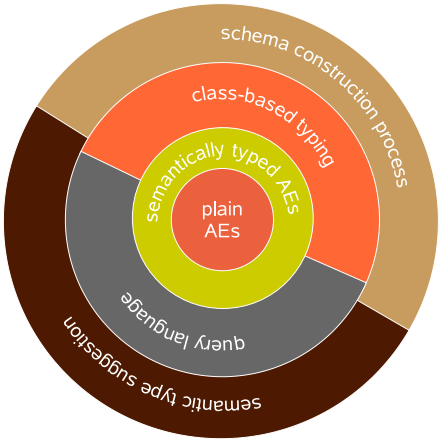


Figure 1.3: *Structure of this thesis around Asset Expressions (AEs).*

1.4 Approach to Solution and Overview

The objectives discussed above will be realized by three contributions:

1. A paradigm called *Asset Expressions* to describe entities medially and conceptually. The language to create Asset Expressions uses a few simple-to-understand paradigms and is easily transformed into a visual representation for the user.

2. Enhancements to Asset Expressions which facilitate the creation of semantic descriptions and the collaboration between users.

3. Processes to work with these descriptions: A process to guide the mapping of description networks onto information systems as well as a process that supports users in working with the descriptions themselves.

Asset Expressions allow users to capture real-world entities by representing them medially as well as providing a conceptual model that again uses medial representations in its explanations. Networks of expressions can thus be created recursively.

Before Asset Expressions are introduced, chapter 2 sets the context for this thesis by reviewing related approaches, particularly in the fields of descriptions of entities as well as of structuring and explaining multimedial content. Models of real-world entities have been built for almost as long as computerized storage of data exists. Some steps of this history are traced in section 2.1. Various applications treat multimedial content as more than just opaque data. Some of their models for content handling are discussed in section 2.3. Asset Expressions reuse bits or gain inspiration from these models.

After this overview, Asset Expressions are introduced in chapter 3. Around their plain version (section 3.1), several layers of extensions are built. These layers are depicted in figure 1.3. Substructuring of content (section 3.2) greatly facilitates the creation of rich models as substructure makes it possible to describe individual parts of the content. Such dedicated descriptions allow the explanation of content by filling the gaps in the knowledge of the audience. Next, Asset Expressions are extended with a semantic type system in section 3.3. The purpose of this type system is to inject some information about the application domain into expressions that can be automatically enforced by a type checker.

Means for efficient use of larger numbers of Asset Expressions in collaborative environments are provided in chapter 4. The semantic typing imposes no structural constraints on the typed expressions. This provides users with great flexibility in modeling their expressions. However, if over-done, structurally diverse expressions can hinder collaboration. Expression *traits* are

introduced in section 4.1 to provide some coupling between semantic types and expression structure. Traits can be thought of as templates, but their use is optional.

If users can share their expressions with others, this audience needs access paths that exceed the pure navigation available in Asset Expressions themselves. To this end, section 4.2 introduces the Asset Expression Query Language (AEQL). There are two use cases for queries: Retrieval of expressions that are interesting to a certain problem and selection of parts of expressions to use in the construction of new expressions. Both cases are addressed by AEQL.

The creation of expressions based on digital multimedial content can best be supported in a dedicated system, as discussed in chapter 5. This system deals with the creation and manipulation of Asset Expressions in general, i.e., without regard to a particular application domain.

The thorough description of application domains in Asset Expressions often leads to a deep understanding of the domain. Regular structures emerge in related expressions. These structures can be exploited to model the application domain in a conceptual model and to then use this model to create an information system which is based on it. A class-based typing of Asset Expressions is provided in chapter 6.

Chapter 7 presents type inference services for both type systems. Semantic types are obtained by matching structural similarities of an expression against previously defined ones in section 7.1. Next, structural types are inferred in section 7.2 from a larger set of expressions to type expressions in the type system presented in section 6. A process is presented to this end to create a conceptual model of the application domain. The process is novel in that it focuses on the domain-expert to create the conceptual model and can usually be run without the help of a modeling expert.

The insights that have lead to the development of Asset Expressions were obtained in practical application projects. Chapter 8 presents one such project and describes how its domain—archival documents from art history—can be modeled using the Asset Expression approach.

This thesis concludes with chapter 9. After a summary of the approach, related work is compared. This includes entity modeling as well as schema construction processes. The chapter concludes with interesting future research objectives.

Chapter 2

Context of This Work

The presented work is related to several contributions either by building on them directly or by carrying over some part of them. These contributions from data modeling, programming languages, and multimedia documents are reviewed in this chapter. The account given of each is not meant to be complete but is deliberately constricted to concepts that are important to this thesis. Literature with more complete coverage is cited. Related work is compared to the approach of this thesis in chapter 9.

2.1 Data Model Extensions

In this section topics on data modeling in a general sense are reviewed. Most contemporary information systems contain a persistent data store. Two models for such stores are outlined next: the relational model and the semi-structured model. Persistent data needs to be interfaced with programming languages to make use of it at the application level. Some approaches to overcome the gaps between paradigms in both worlds are described afterwards.

It is beneficial to create a conceptual model of the application domain as a basis for an information system (IS) in this domain. This model will then influence all parts of the system, including the persistent storage. In particular, it is important that "the IS must model the user's conceptualization of the application domain, not the designer's independent perception, nor should it be a model of the way data is stored in the computer" [Bor85]. Therefore, users should be able to modify the conceptual model according to their needs. This section concludes with an account of the Conceptual Content Management approach that allows such modifications.

2.1.1 Persistent Data

The *relational model* is useful for describing large amounts of regular data. It was first introduced by Codd in 1970 [Cod70]. It is based on the notion of an n-ary *relation*. A relation is the subset of the Cartesian product of the n domains found in the relation. The data in the relation, also called its *extent*, is a set of n-tuples whose values are from the domains specified in the relation. While tuples were originally defined as ordered [Cod70], named attributes in tuples proved more convenient. In ordered tuples, each attribute has to be referred to by its position. This position, however, is defined at the creation time of the relation and is usually rather arbitrary as there is no inherent semantic order of the attributes in most relations. Named attributes offer the advantage of carrying a bit of meaning, which makes them easier to remember. Not relying on an order also is beneficial in processing query results that might contain re-ordered or even combined tuples.

Order(<u>OrderID</u>,Date)
OrderItem(<u>ItemID</u>,OrderID,Name,Quantity,Price)

(a) Two relations

Order	
OrderID	Date
1	01 Jan 1970
2	05 May 1971

OrderItem				
ItemID	OrderID	Name	Quantity	Price
1	1	“Product A”	90	19.90
2	1	“Product B”	1	59.95
3	2	“Product C”	10	0.99
4	2	“Product D”	7	5.99

(b) Some sample data

$$\pi_{Name,Quantity}(\text{OrderItem} \bowtie_{\text{OrderID}=\text{OrderID}} (\pi_{\text{OrderID}}(\sigma_{\text{Date}>30\text{ Jun }1970}\ \text{Order})))$$

(c) A query

Name	Quantity
“Product C”	10
“Product D”	7

(d) The query result

Figure 2.1: *Example data in the relational model with a query that joins both relations.*

All data in a relation are values. This means that the identities of tuples as well as relationships between tuples have to be realized using values. Tuple identity is obtained by introducing a *key* attribute whose value is unique for each tuple in the relation. A common domain for key values are natural numbers. With the help of these key values relationships between tuples can be described by mentioning the identifying key value of the tuple that is referenced in the tuple that contains the reference. This encodes a one-to-one or many-to-one relationship. Other types of relationships can be implemented by additional relationship relations [Amb03].

Queries on relations can be expressed in relational algebra which is used as a formal model of the operations of relational database systems [EN94, chapter 6]. The algebra contains several operators including selection, projection, and join. Selection (σ) is used to obtain the subset of the input set of tuples that fulfill a given selection criteria. Projection (π) obtains a subset—specified by name—of the attributes of all tuples. Join ($\bowtie$) combines tuples from two related relations. The result of the join is the cross product of both relations filtered by the join condition. The results of all operators are again relations. An example is given in figure 2.1.

A widely accepted visualization of relations are tables. For each relation there is one table which is composed of rows and columns such that each row corresponds to a tuple and each column to an attribute of the relation. Tables are also used by the Structured Query Language (SQL, [BS82], most current version [EMK+04, Sta03]). SQL is a declarative language whose query results are based on joins of tables. In analogy to the relational algebra, joins combine tables based on values. The results of joins can be projected to retrieve only the desired column and selection criteria can be given to limit the rows taking part in the join. It is an important property of SQL that all query results are again tables allowing for composition of queries.

The declarative nature of SQL makes it possible for implementations to retrieve the result in the way most efficient for the given dataset and on that particular implementation. This enables a wide range of optimizations. Users of SQL also benefit: They do not have to determine the most efficient access path that leads to the desired result.

While SQL contains means to account for schema evolution by modifications on tables, the corresponding data transformation must be carried out manually. In practice this limits possible evolution steps because it is generally difficult and sometimes impossible to ensure

access to data according to past schemata. Therefore, most applications assume that the schema is defined at the beginning of the application's lifecycle and remains relatively constant afterwards. The creator of a table can also specify data integrity constraints that must hold on the table. These constraints are enforced on all user transactions such that a transaction that would cause the violation of a constraint will fail.

Semi-structured data [Bun97] are data that are not uniform enough to be fully described by a schema but do exhibit some structure. In semi-structured data there is usually structure on a micro scale, e.g., between a few elements of the same document. Abstractly, semi-structured data is represented as a graph of nodes and labeled edges. The graph is usually assumed to be a tree, even though some extensions allow cycles. If cycles are present, there often is a preferred dimension that constitutes a tree.

The reasons for interest in semi-structure data are manifold. Some data are genuinely unstructured. The world-wide-web is an example of this because it contains resources that follow no predefined model and put all sorts of resources into relationships. Another important motivation is that of data integration. This occurs for example when system boundaries have to be bridged and there is no single schema that subsumes the schemata of both systems. Considering the general case of exchanging data between two arbitrary systems, it is apparent that no global schema will be available that subsumes all of them. Data can then be represented as semi-structured data to overcome this lack of conceptualization. The re-interpretation of data according to new—possibly ad-hoc—schemata also is an interesting application. This, e.g., occurs in queries that search for some string of text regardless of the attribute it is contained in (or even including the attribute names themselves).

Data that is hierarchical in nature can easily be represented in semi-structured form. If the same domain offers several possible hierarchizations, one has to be chosen as the dominant one according to which data will be organized. The employees of a business can for example be organized either by physical address or by organizational structure. Whichever hierarchy is manifest in the data will usually receive benefits in querying and manipulation.

A format for semi-structured data that has recently become very popular are languages from the Extensible Markup Language (XML) family. XML offers a data model that is a bit richer than the plain semi-structured graph model by introducing several types of nodes. Some of these nodes can contain primitive values. However, ultimately the model can be mapped onto the plain tree by creating additional nodes and labeling the edges to them with the primitive values. XML is interesting as its applications include all of the motivations for semi-structured data given above: unstructured data (on the world-wide-web in the form of XHTML), data migration (the field of web-services has recently drawn much interest, but many data integration projects based on XML have existed before), and browsing of documents. XML is popular as a storage format for data that is inherently tree-structured.

The existence of *semi*-structured data raises the question of what particular structure can be found in the data. Several means to express such structure exist. An early one are DataGuides [GW97] that collect all sequences of edges found in the data in a single graph. DataGuides can be used to infer a schema that matches the semi-structured data [GW97, GGR+00]. Another language to specify this structure is XML Schema [FW04]. While it seems to be contradictory to specify a schema for semi-structured data, it is important in many applications, e.g., in data integration to determine the exact language that is to be used in an exchange. Schema languages for semi-structured data provide means to specify which parts of the data can be free-form and which have to be rigid.

2.1.2 Persistency in Programming Languages

Persistent data stores offer facilities to store, load, and query for data. The languages used for declaration of data structure as well as those used in query and manipulation are specialized to the task. They are, however, usually not algorithmically complete. This makes it necessary

```
from OrderDB import OrderRel;
transaction OrderExpensive(var O: OrderRelType);
begin
  if all o in O (o.price > 100) then
    OrderRel:+ O
  end
end OrderExpensive;
```

Figure 2.2: *DBPL example allowing only expensive orders with a price over 100. The example uses relation types (`OrderRelType`) and quantified expressions in a user transaction*

to interface with the data store from algorithmically complete programming languages that are used to develop the application which processes the data. Data stores use declarative, set-oriented languages while programming languages are element-based, procedural languages. This difference in paradigms is referred to as the *impedance mismatch*. Database programming languages have three major contributions [Mat93]:

1. *Persistency abstraction.* Transient and persistent objects are treated alike, there are no separate languages for the manipulation of persistent data and transient algorithms but one integrated language. Therefore, there also is no explicit interface to a persistency service in which application programmers would need to cope with the impedance mismatch on their own.

2. *Type completeness.* Mass data—as it occurs in the persistent data store—can be handled by the type system of the language through bulk data types.

3. *Iteration abstraction.* Algorithms for mass data often have iterative qualities which are accommodated by database programming languages through quantified predicates or set-valued expressions.

Integrations of programming languages with data definition and manipulation languages in database programming remedy the impedance mismatch. Realizations of such an integration can be classified in three categories [Mat93]:

1. **Type-oriented database programming languages** are extensions of existing programming languages that are based on a procedural kernel.

   One approach to integrating persistency is augmentation of this kernel with relational types and set-valued operators to treat persistent data [Sch77]. Which data is made persistent is declared by these types. The important concept of database transactions is also present in these languages. Benefits are the integration of facilities to model data and to manipulate it, overcoming the impedance mismatch. An example of this approach is the Database Programming Language DBPL [MS91]. A transaction in this language is shown in figure 2.2.

   Another approach is taken in what is called persistent programming languages. A central aspect for the integration of persistency into these languages is the *persistent heap*. Any object can be stored persistently by placing it on the persistent heap. The scope of persistence is defined by a special object called the *root of persistence*. Any object reachable from the root of persistence is placed in persistent storage—either by static scoping rules or by dynamic binding.

2. **Logic-based and functional database languages.** Logic-based languages achieve persistency by making facts and rules persistent. The model is purely declarative, which

```
[ (Name o) |
    o ← Order;  i ← OrderItem;
    OrderID o = OrderID i; Date o > 30 Jun 1970 ]
```

Figure 2.3: *The query from figure 2.1(c) expressed as a* list comprehension.

gives many opportunities for optimizations. This comes, however, at the price of sacrificing algorithmic completeness. More sophisticated application systems therefore usually require an interface with an algorithmically complete language in order to implement higher-level data manipulations. This carries the danger of introducing a new impedance mismatch [Man91].

The use of functional languages for database programming is appealing because of their closeness to relational algebra. Figure 2.3 shows an example of a query expressed as a *list comprehension* [ART90]. List comprehensions can be transformed into λ-expressions on which optimizations can be executed.

3. **Object-oriented systems** persist objects which are at the same time the entities of transient programming. In this type-completeness they are similar to database programming languages. However, persistency is embedded into the object-oriented paradigm. Data access is often achieved via a combination of quantified, set-valued expressions and path navigation through object trees. An example of this is the Object Query Language (OQL, [Cat94]) which is by itself only a query language, but can be integrated with programming languages to avoid an impedance mismatch [Sub96]. The result of this particular integration is an object-oriented database programming language which is based on relations and in many aspects similar to DBPL.

The LINQ framework integrates query facilities into the language C# [MBB06]. Data that can be queried include objects in the host language, as well as relational and XML data.

Despite the availability of solutions which provide tight integration of data store and programming language and thereby avoid an impedance mismatch, many contemporary information systems are content with keeping data store and application code separate. While this approach has the disadvantages discussed above, there are some reasons for its practicability. The first is the almost orthogonal combinability of data store and application of the programming language of choice. An integrated language will not offer persistency integration for all types of data stores or even their particular implementations. Large systems might contain several data stores at once that function according to different data models. Another reason is the availability of code generators which relieve the programmer of the burden of writing repetitive code that overcomes the impedance mismatch. Also, technologies have emerged that add persistency to purely transient languages either declaratively (for example Hiberate [Ell04]) or by naming convention (for example Ruby on Rails [Cas05]). While still exposing the programmer to an impedance mismatch especially because queries have to be formulated in a different language, practical experience suggests that the integration of data store and programming environment achieved by these languages is smooth enough to be useful.

2.1.3 Assets: Dualistic Description of Entities

The philosopher Ernst Cassirer notes [Cas01] that the pair of content and conceptualized entity go hand-in-hand and neither makes sense in isolation, see section 1.1. Such dualistic descriptions of real-world entities are provided by *Assets*. An Asset describes entities through a medial view as well as a conceptual model view. Both views are specified in Asset classes. The medial view can—as the name suggests—be any medial content. This content serves to the human viewer as a representation of the entity. In the model view the entity is described conceptually.

Following semiotics [Pei31], this description is provided by *characteristics*, *relationships*, and *constraints*, also see section 1.1. A set of Asset classes describing an application domain is combined into an Asset *model*.

Asset classes and models are defined by means of the Asset Definition Language (ADL). Consider the following example of its syntax:

```
model PoliticalIconography
...
class RegentImage {
   content image :  Image
   concept characteristic title :  String
           characteristic epoch :  Epoch
           relationship   regent :  Regent
           relationship   artist :  Artist
           constraint     epoch = artist.epoch
}
...
```

The code fragment defines a model `PoliticalIconography`, which includes a class named `RegentImage`. This class provides in its `content` compartment a reproduction of the modeled image. In the `concept` compartment two characteristic attributes are given. The types of these attributes are taken from a base language, thus the available types and the facilities for creating new ones depend on this language. Assets of the class `RegentImage` can be related to other Assets of types `Regent` and `Artist` through the `regent` and `artist` relationships respectively. The `constraint` forces the epoch of any related artist to match that of the painting.

To enable reuse of Asset classes, existing definitions can be imported when defining a new model. These can then, e.g., be used as base classes to new Asset classes:

```
model RenaissancePoliticalIconography
from PoliticalIconography import RegentImage
class RenaissanceRegentImage refines RegentImage { ...}
```

In addition to the ADL there exist languages for modification and query of Assets as well [Seh04].

2.1.4 Conceptual Content Management

Conceptual Content Management (CCM) uses Assets to provide descriptions of real-world entities through pairs of medial content and conceptual abstractions as requested by Cassirer. A detailed account of the approach can be found in [Seh04]. CCM provides a conceptual model of the domain in order to generate an application system from it. CCM overlaps with traditional content management [Tri05] in also separating the description of entities (in documents in content management) from their presentation. Dualistic descriptions, however, are not found in traditional content management. CCM has been applied to a variety of topics [SSW01, SBS05, BSSS05, BSS05].

Most contemporary software engineering processes encompass an analysis of the application domain at some point. Based on the results of this analysis—which are often expressed in a conceptual model—an information system can be built which supports various tasks found in the application at hand. However, the underlying assumption that the whole system can be based on *one single* conceptual model is not always true. Moreover, it is usually also assumed that the conceptual model does not change.

It can be observed in many application domains that users of an information system hold heterogenous views on the domain. These views can differ in a variety of ways. Users can be interested in a more detailed model in some specific area that is of particular interest to them.

This area may of course be different for any two users. Also, users can disagree on what should be included in the application domain, i.e., what is of interest in general to the application at hand. One user's primary interest might be another user's related work. In a direct clash, users can hold different opinions of what is an appropriate model of the application domain. If communication is possible at all between such users, these differences are likely to rather be in the details than in the broad picture.

The Asset definition language allows users to express their personal opinions in the conceptual model. They can personalize existing definitions based on the common model:

```
model MyPoliticalIconography
from PoliticalIconography import RegentImage
class RegentImage {
   content description:  MP3File
}
```

This adds a content to the model `MyPoliticalIconography` which is personal to the user. Any attributes of `RegentImage` which are not mentioned remain the same as in the original. Users are not limited to adding attributes or classes but can also modify all other aspects such as types or cardinalities of relationships [Seh04].

This ability of users to modify the conceptual model is referred to as *open modeling*. Not only can the conceptual model be changed at all, but users (or groups of users) are able to do so on a personal basis. This can potentionally result in several parallel models. Traditional information systems technology is neither able to cope with several models at a time, nor can it react on its own to user changes to the model. The latter is necessary to make open modeling practical by automatic adjustment of the system to the new specification.

Several other approaches towards personalizable systems exist, e.g., [Sch99, Rie00, RSG01]. However, these approaches realize personalization at runtime through generic implementations. Users are therefore not able to express their view of the application domain in a personal conceptual model.

When several users of the system can all model their own personal opinion in a conceptual model, it is unrealistic to assume that each of these models can be manually implemented by an application developer. Moreover, changes to models might be frequent, but open modeling is only feasible for users if the new system (implementing their new model) is available to them in a timely manner. Both arguments rule out the traditional approach of manual implementation. Instead Conceptual Content Management Systems (CCMSs) must be able to adapt to new models on their own and in close to real time. This ability of systems is referred to as system *dynamics*. System dynamics and model openess go hand-in-hand.

System dynamics can be achieved by means of a compiler framework and a modularized architecture. The compiler framework uses the Asset model to create a full CCMS which conforms to this model. The CCMS is composed of components which in turn are broken down into modules. The compiler framework uses several *generators* each of which creates a particular module. A particular system is created by a run of the compiler framework with an appropriate configuration of generators. Details on the compiler framework can be found in [SBS06].

Modules are self-contained units which form the smallest building block of a CCMS. There are different kinds of modules. Central to the notion of a module is the uniform module interface which is used for communication between modules. This interface provides means to create, modify, and delete Assets as well as to query for existing ones. All operations are available in different forms, e.g., to also deal with sets of Assets.

The operations of the module interface are common to all kinds of modules, even across components. All modules of a component deal with Assets of a fixed model (according to the domain of the component) through this common interface. Modules therefore are freely combinable while still being domain specific (as only Assets from the model of the component

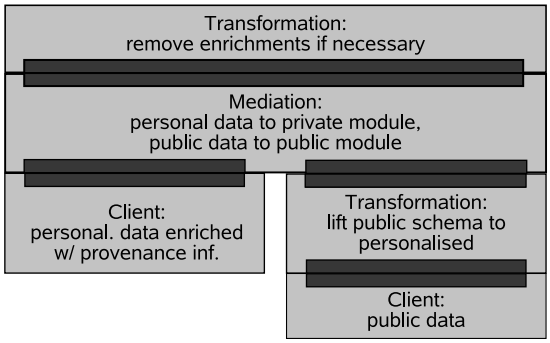


Figure 2.4: *Module setup for personalization. Additional provenance information is introduced to track personalized variants of public Assets.*

are available through this interface). Their combination is realized in layers where modules from higher layers use one or several modules on the layer below. The dependencies of modules form a directed, acyclic graph. The most important types of modules are discussed below.

Client Modules. Client modules provide the uniform module interface to modules on higher layers. Client modules do not use any further modules but map all calls they receive onto some third party systems. The most common case of third party system is a database used to store Asset instances.

Other client modules might forward calls to a remote system that uses a communication mechanism different from the module interface. Such remote systems can be outside the CCMS entirely, for example a system accessible via a web service.

Server Modules. Server modules are complementary to client modules. They use modules on the layer below them via the standard module interface but do not provide this interface at all. Instead, they usually provide some other interface, but options for this interface are very diverse. They include interfaces for human users to the CCMS—for example a web application or a fat client—or the remote side of a standardized distribution mechanism (such as a web service or CORBA[1]).

Transformation Modules. Transformation modules both use and provide the standard module interface. They are based on exactly one module on the layer below them. There are three types of transformations:

1. Temporal Transformation modules make use of a client module to make Asset instances long-lived. This is the persistence mechanism of CCMSs.

2. Spatial Transformation modules are used to achieve physical distribution between systems. Spatial transformation modules are logical modules as they are part of two different systems at the same time. To this end they are composed of two modules: a client and a server module, which use some distribution mechanism between them.

3. Schema Transformation modules use and provide the standard module interface. However, in the case of schema transformation modules both interfaces conform to different schemata.

Mediation Modules. To provide access to different information sources in a homogeneous way, mediators can be used [Wie92]. Mediators themselves include a wrapper around the information sources as well as some mediation logic to tie them together. This mediation logic is implemented by mediation modules, the wrapper by transformation modules. Types of mediation modules differ in their mapping of module operations to their base modules:

- Schema variant mediation modules are based on two underlying modules and used in cases where there is a different schema on one of the base modules. Instances are read from both base modules, creation and modification are done in the module for the new schema, and deletion happens in both modules. A component implementing this scenario is shown in figure 2.4.

- Instance personalization modules combine a module with publicly available Assets and a module with Assets personal to the user of the module. The user can modify instances according to personal opinion without interfering with the instances defined in the public module. This mediation module is based on a public (for access to the data of the system) and a personal module (for the user's modifications). Instances are read from both modules with personalized variants hiding public instances, creation is done in the personal modules, modification is done on a personal copy of the public instance, and deletions are recorded in the personal module.

- Hub modules multiplex calls to their interface to a number of underlying modules. All base modules have the same role.

Complete Conceptual Content Management Systems are a setup of a large number of modules. A great variety of functionalities can be realized by module combination. Figure 8.6 shows a real-world system which exhibits several ways of module combination.

2.2 Foundations

2.2.1 Functional Programming

In mathematics, a given variable usually refers to the same value throughout a calculation. Otherwise it would make little sense to talk about the solution of an equation such as $x^2+1 = 2x$. In programming, this is not always the case. In a procedural language, such as Pascal, one can write assignments, e.g.:

$$x = x + 1;$$

It is understood that both occurrences of x refer to different values. In a mathematical context, one would distinguish both occurrences by different names. More precisely, when seeking the solutions to an equation

$$\exists x,\ x^2 + 1 = 2x$$

the variable x is said to be bound to some—possibly not yet known—value as opposed to a free variable.

The λ-calculus offers a possibility to make explicit which variables are bound and what the scope of their binding is in a functional program. It uses a special binding symbol "λ" that identifies bound variables occurring in an expression:

$$\lambda x.x + 1$$

[1] CORBA: Common Object Request Broker Architecture [OMG96]

The λ-calculus is a simple, yet well-understood paradigm to express functional programs. In its purest form, it only contains three concepts: variables, abstractions of variables from expressions, and applications to expressions. However, constants are usually also introduced for brevity. The syntax is [Rev88]:

$$\begin{aligned} &\textit{expression} ::= \textit{variable} \mid \textit{constant} \mid \textit{application} \mid \textit{abstraction} \\ &\textit{application} ::= (\textit{expression})\ \textit{expression} \\ &\textit{abstraction} ::= \lambda \textit{variable}.\textit{expression} \end{aligned}$$

Variables are names, usually single characters. Constants are literal values of the domain of allowed constants. The variable in abstractions is called *abstraction variable*. The first expression in an application is its *operator*, the second its *operand*. Thus some examples of valid expressions are:

$$\lambda x.x \qquad (x)y \qquad (\lambda x.x)y \qquad \lambda x.3$$

Many authors also include some standard operations. Both constants and operations can be expressed in the pure version. The extensions are therefore only syntactic sugar to improve readability of common expressions:

$$\lambda x.\lambda y.((+)x)y \qquad ((\lambda x.\lambda y.((*)x)y)3)4$$

The first expression is a function that will perform an addition, the second an instance of multiplication that would equal 12 if executed.

When combining expressions, one sometimes needs to avoid clashes in naming. This can be achieved by renaming the clashing variables. The renaming of a variable x to z in an expression E is written as

$$\{z/x\}E$$

The resulting expression can be obtained by recursing on the structure of E and renaming all encountered variables x and abstractions λx to z and λz, respectively. For a formal definition see [Rev88, chapter 2.2].

The substitution of an expression Q for all free occurrences of the variable x in P, i.e., $[Q/x]P$ can then be defined inductively as follows [Rev88]:

1. $[Q/x]x \cong Q$
2. $[Q/x]y \cong y$, if $x \neq y$
3. $[Q/x]\lambda x.E \cong \lambda x.E$
4. $[Q/x]\lambda y.E \cong \lambda y.[Q/x]E$, if $x \neq y$ and (x not free in E or y not free in Q)
5. $[Q/x]\lambda y.E \cong \lambda z.[Q/x]\{z/y\}E$, for any E and for any z such that $x \neq y \neq z$ and z does not occur in $(E)Q$, if $x \neq y$ and (x free in E or y free in Q)
6. $[Q/x]E)F \cong ([Q/x]E)[Q/x]F$

Rule 5 is essentially the same as rule 4, but it avoids the capture of the y that occurs in Q by renaming it to z. Using this substitution, the β-rule defines simplification of expressions:

$$(\lambda x.E)F \rightarrow [E/x]F$$

Based on the β-rule, reduction of expressions can be defined, which makes it possible to express calculations. Reduction can occur when a parameter is applied to an expression in which a variable is bound by abstraction. Thus, the value of the multiplication in the example above can be obtained as:

$$\text{Abstraction} \quad \frac{\Gamma, x : \tau \vdash E : \sigma}{\lambda x : \tau . E : \tau \to \sigma} \tag{2.1a}$$

$$\text{Application} \quad \frac{\Gamma \vdash E : \sigma \to \tau \qquad \Gamma \vdash F : \sigma}{\Gamma \vdash (E)F : \tau} \tag{2.1b}$$

$$\text{Variable} \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \tag{2.1c}$$

Figure 2.5: *Typing rules of the simply typed λ-calculus. Γ is the typing context which holds a sequence of variables and their types.*

$$((\lambda x.\lambda y.((*)x)y)3)4 = (\lambda y.((*)4)y)3 = ((*)4)3 = 12$$

The last step requires the expansion of the syntactic sugar for multiplication and constants.

Expressions in the λ-calculus can be typed with what is called a *simple typing* [Pie02]. The idea is to assign each variable a type. The type of an abstraction is then a function type whose domain is the type of the abstracted variable and whose range is the type of the expression that is abstracted from. Applications are only well-typed if the types of the operand and the operator match. Intuitively, this means that the operand must be a proper argument to the operator.

There are some primitive types to accommodate the built-in constants, such as int and bool. Function types are obtained by the type constructor →. A function of type $\sigma \to \tau$ therefore has domain σ and range τ. The type of an expression is separated from the expression by a colon.

The function that always adds 2 to its argument

$$\lambda x : \text{int}.((+)x)2$$

is of type int→int. The type assigned to an abstraction variable might be redundant as it is sometimes possible to infer types from the expression. In the function above for example, x must be of type int as the function + is of type int→int. This allows for typechecking against the specified types. The expression

$$(\lambda x : \text{int}.((+)x)2)10$$

is of type int. Any applied operand that is not of type int would make the expression ill-typed:

$$(\lambda x : \text{int}.((+)x)2)\text{true} \qquad \textit{; type error!}$$

Figure 2.5 summarizes the typing rules (a summary of the syntax of type inferences is given in section 3.3.4 on page 63 for readers who are unfamiliar with it). The *typing context* Γ holds a set of variables and their types. The comma operator (used in the typing rule for abstraction) appends a type assignment to the context. Only applications whose operator contains an appropriate abstraction are well-typed as expressed in rule 2.1b. A variable is of type τ if it is defined as such in the typing context.

2.2.2 Denotational Semantics

The meaning of programming languages can be defined by denotational semantics [Sto77, All87]. Semantic valuation functions map syntactic constructs in the program to values, for example numbers or booleans. The semantics of a whole program can be obtained through the recursive definition of the semantic valuation function. This function is defined in such

a way that the value of a larger syntactic construct is defined in terms of the values of its sub-constructs. This division must eventually arrive at atomic units that have no further subdivision if the semantics of the whole are to be obtained.

Binary numerals num are usually written with an alphabet of $\{0, 1\}$ and have the syntax

$$\nu ::= \nu\delta|\delta$$
$$\delta ::= 0|1$$

The semantics of binary numerals, that is the integers int they stand for, can be expressed by a semantic valuation function $\mathbf{V}$:num→int

$$\mathbf{V}[\![0]\!] = \mathtt{0}$$
$$\mathbf{V}[\![1]\!] = \mathtt{1}$$
$$\mathbf{V}[\![\nu\delta]\!] = \mathtt{2} \times \mathbf{V}[\![\nu]\!] + \mathbf{V}[\![\delta]\!]$$

Of course, digits in typewriter font $\mathtt{0}, \mathtt{1}$ represent semantic objects, but writing down the valuation function makes it necessary to adopt some syntax for semantic objects. The first two lines define the semantics of atomic syntactical objects that are not reduced further. The third line serves to decompose a complex construct into two simpler terms by separating the last digit from the numeral.

In a similar manner as the above example, denotational semantics can be given for more complex languages than numerals. The approach is always the same: Subdivision of large constructs into smaller ones until atomic expressions are reached that can be directly translated. To humans the appeal of semantics defined denotationally depends on the intuitiveness of the translations of the atomic syntactical elements and on the approach taken to decomposition of complex elements.

2.2.3 Ontologies

Ontologies have become a popular research topic in several fields because they aim to solve a rather common problem: Ontologies provide a common understanding of an application domain. They are used to formalize this understanding in order to distribute it among all concerned partners. Despite their popularity, there is some disagreement as to what exactly an ontology is and how it should be specified [NH97, PGC02]. A definition that is general enough to be shared by most contributors to the field is the frequently cited ontology as "a formal, specific conceptualization of a domain" [Gru93]. The term "ontology" is borrowed from philosophy where it refers to the study of the nature of being. In this sense it is usually spelled with a capital "O" to distinguish it from its use as a system of categories. Ontologies are used for a variety of purposes and in several different meanings [May02]: as schemata or as meta-models, to explicitly define the meaning of vocabularies used in natural language, as a "core body of knowledge" to be used as a basis for further developments, as well as their original philosophical sense.

Ontology Definition

It is generally agreed that an ontology captures the conceptual structure of a domain [MS01]. In a simple form it can be given as a number of terms (the lexical references) which are then associated with entities from the domain of discourse. Many ontologies also provide classes which capture the conceptual structures of the application domain. These classes are defined intensionally by attributes characteristic to them, which separate them from other classes.

McGuinness [McG03] lists possible candidates for ontologies shown in the ontology spectrum in figure 2.6, beginning with the very simple notion of a controlled vocabulary. More complex notions are thesauri—e.g., WordNet [Fel98]—or systems with an informal "is-a" relation. Other authors also consider database schemata a form of ontology [KS03]. McGuinness considers

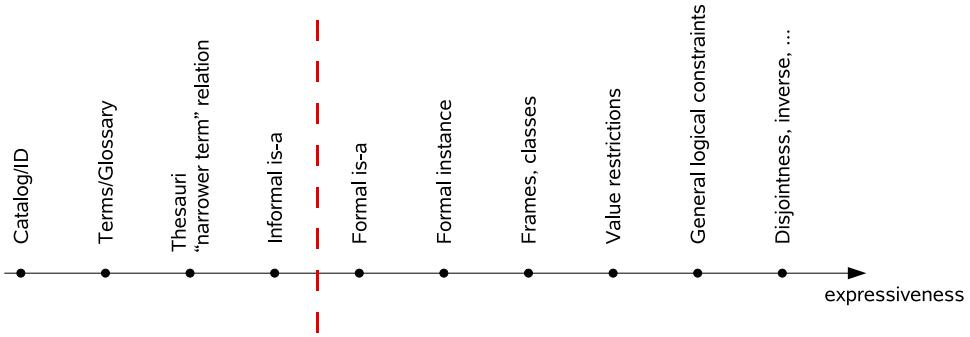


Figure 2.6: *Spectrum of ontologies after [McG03]. The dashed line marks least expressiveness for a paradigm to be commonly considered an ontology, though this border is no a general consensus.*

all paradigms in the spectrum that have at least a formal "is-a" relation to be ontologies. More expressiveness is gained by extension of the formalism with structural classes, different kinds of constraints, and more complex notions such as disjointness or inverses. Conceptual modeling approaches such as CCM (section 2.1.4) have an expressiveness that is approximately found between "Value restrictions" and "General logical constraints". The dimensions "model acceptance" and "model scope" can be added to that of "expressiveness" to classify ontological models and their particular implementations [SGW05].

While ontologies share some properties with database schemata—such as the structural definition of classes—there are important differences. An ontology is used for collaboration and must therefore be a shared understanding. Furthermore, an ontology provides a description of a domain, not an implementational view of a particular part of this domain in a computer system. Conceptual schemata [BMS84] are more similar to ontologies as they also aim at describing the domain, not the implementation.

Many applications of ontologies use a paradigm in which the definitions in the ontology have structural influence on the descriptions of instances in the application domain. These paradigms mostly differ in the formalism used to express the ontology. The following definition is proposed by Maedche [MS01]: An ontology is a quadruple

$$O = \{C, R, H^C, A^O\}$$

including a set of concepts C, a set of binary relations $R \subseteq C \times C$ of concepts, a concept hierarchy $H^C \subseteq C \times C$, and a set of axioms on the ontology called A^O. This ontology is populated with concepts and relations between them. However, it still is difficult to talk about elements of this ontology as there are no names for concepts nor relations. To remedy this, a lexicon is introduced. The lexicon provides names of concepts and relations. It also contains translations of those names to concepts and relationships.

Ontology Creation and Evolution

The extent to which a domain is modeled depends on the preferences of the ontology designer. Five design criteria have been suggested for ontologies [Gru93]:

- *Clarity:* The ontology should clearly communicate the intended meaning of its concepts. To this end it is important to be objective in the definitions even if the reason for including the concept might have been subjective. Formalization is mentioned as a means towards achieving objectivity. In addition to formal definitions, natural language definitions should also be provided for all concepts.

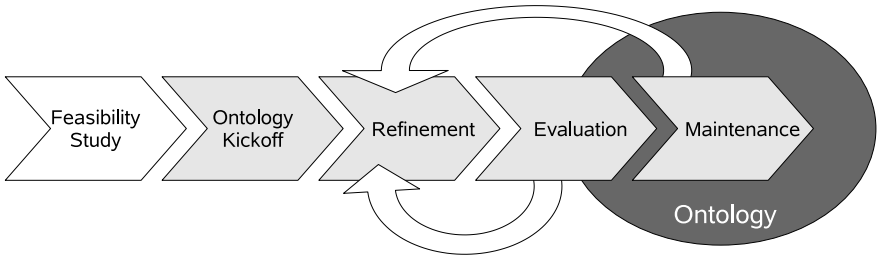


Figure 2.7: *The On-To-Knowledge meta-process [SSSS01].*

- *Coherence:* The ontology should be consistent. This should also cover the concepts which are defined informally.
- *Extendibility:* The ontology should provide the foundations for anticipated tasks. It should be possible to later define new concepts based on the existing ones.
- *Minimal coding bias:* Representation choices should not be made for convenience of expression with a certain formalism.
- *Minimal ontological commitment:* The intended knowledge sharing should be made possible with as few ontological claims as possible. In particular, only those terms that are necessary for the communication should be defined.

These design criteria overlap with quality measures for conceptual models [CACW02]. Especially the first three criteria of clarity, coherence, and extendibility are also discussed with respect to conceptual models [Wed00]. However, in any ontology—just as in a conceptual model—there will always be a certain measure of subjectivity. [Mus98] notes about ontologies:

> "They do not, and cannot, capture absolute Platonic truths about what might exist in the world. The merits of a particular ontology can be measured only in terms of how well that ontology supports development of the application programs for which it was designed, and of how easy it is for developers to reuse that ontology to build new applications."

In practical projects ontologies are sometimes simple taxonomies [SSS04] or are used as conceptual models [SMJ02]. Taxonomies provide "is-a" relationships between their concepts, which is a notion whose formality can still be coped with by some domain experts [GW02, SSS04].

Regardless of the level of formality that the ontology is expressed with, the creation of the ontology is usually carried out by an expert who does not only have an understanding of the application domain but is also familiar with the modeling concepts. If ontologies are used in the development of information systems, some process support for their creation as well as the integration of such a process with the overall application development is necessary [Gua98].

Figure 2.7 shows the On-To-Knowledge meta-process [SSSS01], which can be used to systematize the creation of ontologies. It is part of a process for the development of knowledge management applications (CommonKADS, [SAA99]). The On-To-Knowledge process consists of five phases: (1) Feasibility study, (2) ontology kickoff, (3) ontology refinement, (4) evaluation, and (5) maintenance. It contains two iterative elements: It is possible to revert to the ontology refinement phase from both the evaluation as well as the maintenance phase. The first iteration accounts for any errors that might have been made in the development of the ontology, the second takes care of changes to the ontology that are necessary throughout the lifetime of the project. During ontology kickoff the requirements to the ontology are specified,

and input sources (such as reuseable ontologies, domain experts, use cases, etc.) are identified. A semi-formal ontology is created. The refinement phase is for concept elicitation with domain experts and the formalization of these concepts into a fully formal ontology. The evaluation phase checks the requirements to the ontology and tests the ontology in its target environment that is build in the overall application development. The ontology is applied to the knowledge management system in the maintenance phase.

A process that is specifically aimed at the evolution of ontologies to cope with changed requirements is presented in [SMMS02]. A central goal of this process is to ensure that consistency is maintained across changes to the ontology. Furthermore, the user is assisted by making modifications of ontologies easy to carry out and by offering advice on further ontological refinement. This is achieved in four stages, which formalize the whole process from the request of an ontology change all the way to the propagation of the change (and its consequences) to dependent ontologies and applications.

Ontology Applications

Once defined, an ontology can be used in many ways. An overview is provided in [McG03], including:

- *Consistency checking*: Ontologies can be used to assess the consistency of information provided by the users or other sources outside the system. The constraints defined in the ontology facilitate the early detection of errors in such information. Common examples include value restrictions and type information. Similar functionality can be realized by constraints in conceptual models. The difference between the two with regard to consistency checking is mainly in the point of view: The ontology is not used as a direct foundation of the information system and thus does not take the system's point of view.

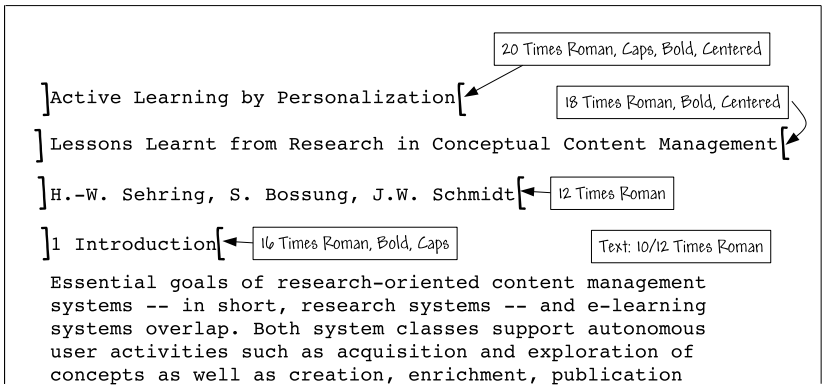
- *Interoperability support*: Even with simple ontologies such as taxonomies or controlled vocabularies, there are benefits for interoperability since all involved parties can use a common set of terms to communicate. More expressive ontologies can even specify the entire language of communication by giving structural definitions for the terms in the form of classes. This use of ontologies is often found in information integration tasks, e.g., of XML documents [ES00]. A survey of mapping approaches through ontologies is provided in [KS03].

- *Supporting structured, comparative, and customized search*: The effectiveness of queries largely depends on the user knowing what to ask for. Ontologies can help in two ways: By providing terms from the application domain along with synonyms, antonyms, etc. and by offering properties of those terms (if the ontology is expressive enough) by which the user can further refine the search.

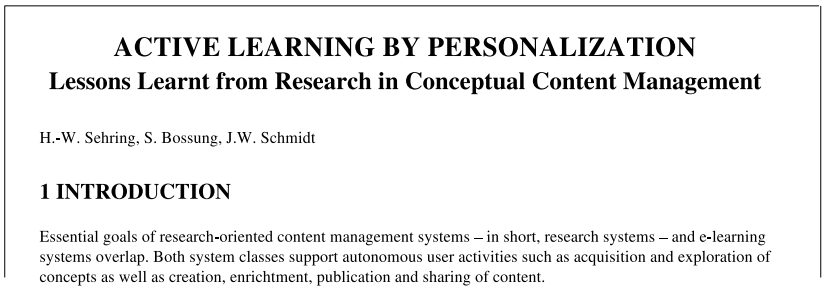
- *Exploiting generalization/specialization information*: A too general query might result in a large number of answers. This number can be reduced if the terms in the query can be refined by means of the ontology. A search engine that can discover concepts from the ontology that correspond to the query terms can even suggest useful refinements of the query that lead to a managable number of answers. Whole retrieval models can be based on ontologies [VFC05].

2.3 Semantic Models for Multimedial Content

In the following, an overview of paradigms for the handling of multimedia documents is given. It is broken up into three parts: Paradigms that structure individual documents, hypermedia systems that focus on the relations between documents, and approaches that also provide semantic descriptions of multimedia documents.



(a)



(b)

Figure 2.8: *A document marked up for typesetting to convey information which the typewriter cannot produce directly (top) and the document rendered according to the information provided in the markup (bottom). This constitutes specific markup.*

2.3.1 Structured Documents

Two ways to structure documents are discussed below: markup and annotations. This discussion is by no means exhaustive, not in the depth of the topics but especially not with regard to discussing all approaches. There are for example also paradigms such as superimposed information [DM99] (consolidating information from several sources), which are similar to markup and to annotations. However, a detailed account of them is not necessary for the later discussion of the Asset Expression approach.

Markup

Enriching documents with markup has been common practice in many fields, long before digitized documents were available as a basis. Markup is additional information which is added to a document for example to make its structure explicit or to describe its formatting. Markup is generally added directly into the text. In the words of [Wat02] "document markup is the process of adding codes to a document to identify the structure of a document or the format in which it is to appear." Heavy use of markup occurs with people who work with texts professionally: authors, copy editors, and typesetters. Figure 2.8 shows an example of such markup, which describes the exact formatting of a text in a document written on a typewriter. As the typewriter cannot produce the desired formatting directly, particular codes to add this formatting information are inserted into the text. The formatting of web pages is specified in a similar way today.

Just as in the paper-and-pencil case, storing text with markup is what most electronic word processing tools do. The actual text of a document is enriched by special codes which store the formatting of the text. In text processing tools the markup is usually hidden from the user who is presented a rendered view of the effect of the markup. As such markup is internal to the word processor, it is called *specific markup* in [Wat02] (or *procedural* markup in [CRD87]). Specific markup contains precise instructions to its audience (e.g., the word processing tool which created it) but is generally difficult or impossible to understand for other audiences (e.g., humans[2]). Contrary to specific markup, *generic markup* describes the structure of a document. Generic markup is sometimes also referred to as *descriptive markup* [CRD87] as it tries to describe the role of a part in the document instead of presenting instructions on what to do with this part.

While specific markup might mark the headline of an article as "bold, centered, 18pt", generic markup might simply mark it as "heading-1". The information on how to render the document is stored separately in the case of generic markup. Instead of providing a style directly, generic markup builds a document from a set of logical components, which are assumed to be formatted consistently. A word processor can for instance be instructed to format parts marked up as "heading-1" in "bold, centered, 18pt", but the generic markup makes sense to a wider audience than just this specific word processor. The processing system Latex [Lam86], which was used to format this text, largely constitutes an example of generic markup: Text is structured explicitly, but the formatting of these structures is left to a document style sheet.

With many communities interested in text markup, the adoption of a standard format became important to facilitate dealing with marked up data, for example in exchanging it between tools. This lead to the creation of the Standard General Markup Language (SGML, see [Org86]) in 1985. SGML is aimed at all document authors who can use whatever markup they desire. Such high flexibility was deemed crucial by the creators of SGML as it allows users to invent new markup for particular application domains or to ensure that the markup can be done in the native language of the participants [Gol90]. SGML is therefore not one particular language, but rather a family of languages. For an application domain, a member of this family is defined and used in instance documents. SGML thus combines broad coverage of domains with high flexibility. Some feel that this makes SGML cumbersome and difficult to learn [Pri96]. As a successor of SGML, one of the goals of XML [BPSM+06] was to somewhat remedy this situation by specifying a family of languages which is easier to parse and process and includes better internationalization. However, human readability is still not achieved, which is witnessed by very few applications outside the domain of programming that use XML directly as an interface to the user. Both SGML and XML partially standardize the syntax of markup to facilitate processing. Means to explicitly connect the markup to its semantics still need to be provided and a standard is missing in this area [RDSM02, RDSMH03].

Markup—both generic and specialized—is not haphazardly inserted into text but used to convey or amplify some meaning. Depending on whether the markup is inserted by the original author or is a result of a later interpretation of the text, markup can be classified into *constitutive* or *interpretative*, respectively [SMHR03]. While the former expresses the author's intentions, the latter is the opinion of a third person. It is plausible to imagine disputes about the correct interpretative markup, especially in absence of constitutive markup. [Sim94] uses different visual models (renderings of marked-up text) over the same "conceptual model" (marked-up text) to settle such disputes. One application is to build consensus via two such views between researchers with different areas of interest. [SMHR03] create a formal model of markup as nodes of a document tree. This model aims at making inferences about the document based on the markup. The authors point out that while they make their arguments with regard to marked up text, the same arguments apply mutatis mutandis to other material,

[2]The reader is encouraged to try this with a document from the preferred word processor. Even recent XML-based office formats [DBO06, Mic06] are essentially illegible to humans. Whether they constitute specific markup generally depends on the way the word processor is used.

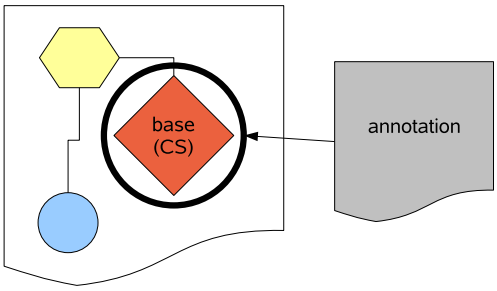


Figure 2.9: *An annotation is connected to its base, a characteristic structure (CS) in the document. This pair constitutes a characteristic pattern (CP).*

e.g., records in a database. Central to the model is the definition of properties, which are attached to elements of the document. Inferences are then made over these properties, but the properties themselves are not further defined or explained.

Document markup is information which is conveyed by codes directly inside the document. Markup is often created by the author of the document at creation time. However, in the case of interpretative markup it may be inserted later.

Annotations

Annotations follow a similar path as interpretative markup: They are added to the document a posteriori. In fact, some authors consider markup a form of annotation [OAM99]. However, unlike interpretative markup, annotations do not reside in the document itself. Instead, they reside in a separate environment and are linked to the document. A common purpose of annotations is to clarify some aspect of a document by making further information on this aspect available.

Humans interpret documents by recognizing sets of elementary signs in functional or perceptual units. These signs are called *characteristic structures* (or CS for short) [BCM99]. CSs can take many shapes such as letters of the alphabet or icons in technical drawings. CSs are associated with a meaning. Such associations are called *characteristic patterns* (CP). Humans combine elementary CSs into complex ones, such as words in a textual document or machine parts in a technical drawing. These complex CSs give rise to complex CPs, enabling the whole document to be recursively recognized.

An annotation is a note [FFM04] which is attached to a (possibly complex) CS. The CS it is attached to is called the *base* of the annotation. In principle, the note itself can take any format, but it is usually restricted for various reasons in concrete cases. As the annotation explains the meaning of its base, the pair of base and annotation constitute a CP. This relationship is illustrated in figure 2.9. The base of the annotation is often marked by a special identifier such as a curve around a region. The link between the annotation and its base is often also expressed visually to created a stronger connection of the pair, for example by drawing an arrow.

The previous examples are based on documents with a physical support, e.g., texts printed on paper. Opportunities to work with annotations have greatly increased since more and more documents became available electronically. When a document is only available on a physical support, this limits the amount of annotation which can be attached directly to the document simply because there is limited space available. Physical annotations can also be made in separate locations, such as other sheets of paper. Creating a link between the annotation and its base can then become challenging. Moreover, this link is more difficult to follow for the reader of the annotation.

Electronic annotation can take a variety of forms. This variety makes an issue of the

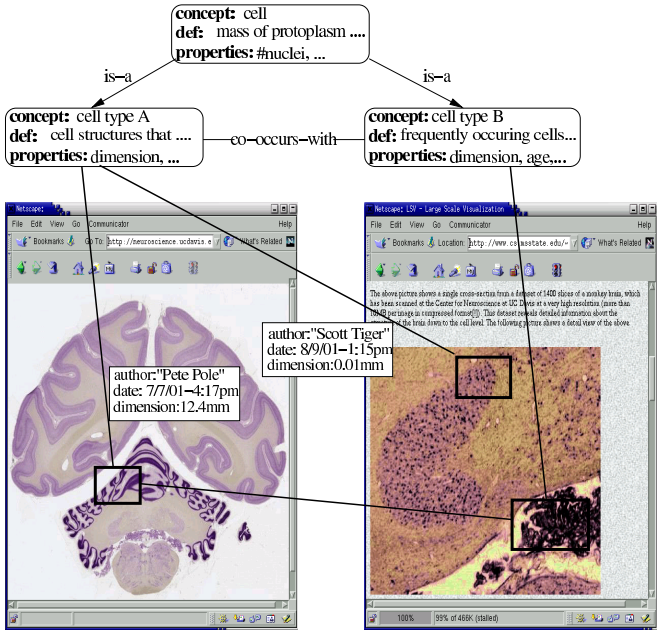


Figure 2.10: *Conceptual annotations use a controlled vocabulary to enable shared understanding. Figure from [GSG+ 02].*

availability of the appropriate technology to render the annotation itself as well as the link to its base. Annotations to documents on paper have a built-in, if not obvious, support for both [SR03]. However, with computer-based examples, access to the annotation depends on access to appropriate technologies. In the electronic version, the annotation might be stored separately from the base document, making annotations more flexible.

Due to the diverse nature of electronic annotations, it is somewhat difficult to define what exactly constitutes such an annotation. While there does not seem to be general agreement, most researchers could agree to some of the following points (not all of which must be met by every application) [SR03]:

- An annotation must be attached or linked to a base document in such a way that it can be retrieved where appropriate. This might require the use of some specific technology.
- An annotation is viewed alongside the original object.
- An annotation is created after the creation of the base document.
- An annotation does not modify the base.
- The distinction between annotation and document data can be fuzzy and depend on the activity at hand.
- Some annotations require particular knowledge to decode their meaning.

When the authors of annotations are free to use any language or other description means they want, this creates a problem of semantic heterogeneity, where it is potentially difficult to understand an annotation created by somebody else [KS98]. However, annotations can be used to associate agreed-upon metadata with diverse data sources [GS01, GS02]. Figure 2.10 shows several examples of such conceptual annotations. In this approach, the key idea is to avoid free-form annotations in favor of more structured descriptions, possibly even controlled

by a shared ontology. Documents that are annotated by means of such a shared ontology can then be integrated via the shared concepts [GSG+02]. A taxonomy of annotation links can be used to formalize the relationship of an annotation and its base [AFO05].

Improved access facilities to medial data can be provided through annotations on the media. Most research on such augmented access facilities is done in the area of images. One such approach is the annotation of images with keywords. The authors of [BDF+03] present a model that breaks images down into regions (CSs) and assigns words to these regions thus providing a simple form of CPs. It is assumed that the words are self-explanatory to the target audience. The regions are computed automatically and assigned with a number of computed features which pre-iconographically describe the region. Based on these features, words are assigned to the region. The approach of [BVNK01] is similar to this as it is also uses image regions (called "spots") to allow users to pose queries against a collection. Textual annotations over images are also used to create captions for the images [HPS99]. From images with complex annotations, captions describing particular aspects of the image (such as which parts of an object are visible) can be created. The approach is geared towards addition of captions to graphics in interactive systems. It thus works with a limited set of images of which several variants exist. Automatic annotation of web pages is also subject of research [MYR03]. All automatic annotation approaches aim to recognize some features in the base documents and to then attach annotations built from a limited vocabulary. They therefore implement some form of conceptual annotations, though the degree of formality of the vocabulary differs greatly. At the one end of the scale, the vocabulary is assumed to make sense to the reader as plain language, at the other end, it is backed by precise ontological definitions.

Further annotation systems exist for a variety of content. The systems presented in [PW97, NSS01] deal with a variety of document formats. Based on the observation that content is inaccessible without alternative description [NSS01], the authors propose the idea of a web superstructure which is built from layers of content and metacontent. The meta-layer allows authors to provide external annotations to resources.

In all discussed approaches, the annotations inhabit an environment different from that of the base documents. In some cases, the documents and the annotations are restricted to different kinds of media (e.g., [BDF+03, BVNK01]), in others the two are stored in explicitly different layers ([NSS01]). Therefore, the process of annotating is not reflexive, in the sense that a document could be an annotation and the process would stay in the same domain.

2.3.2 Hypermedia

Hypermedia systems interrelate medial content by creating links between these contents commonly called *documents*. This leads to a graph in which nodes are documents and the links are represented as edges. The functionality of hypermedia systems differs greatly in terms of supported media types, granularity of links, and the sophistication of display of hypermedia structures. The granularity of linking refers to the base of links. Some approaches use whole documents as bases, calling for a rather fine-grained division of documents if precise link bases are desired. Others can use fragments of documents as link bases, for example single words in hypertext systems. Hypertext systems in the strict sense only deal with plain text, but many have been augmented to include other types of content, eventually resulting in hypermedia systems.

The idea of hypermedia systems as a means that greatly facilitates working with large amounts of structured content dates back to 1945. In his paper "As We May Think" [Bus45] Vannevar Bush states the problem of discovery of information in weakly linked structures (e.g., libraries which sort alphabetically or by some key), requiring repeated drill-downs along some ordering scheme to obtain related information. He observes that selection of information in the human mind works much differently: humans associate related information, there is no global indexing scheme. To overcome the mismatch in the way information is structured, he proposes a machine that stores vast amounts of content and provides associative indexing for it.

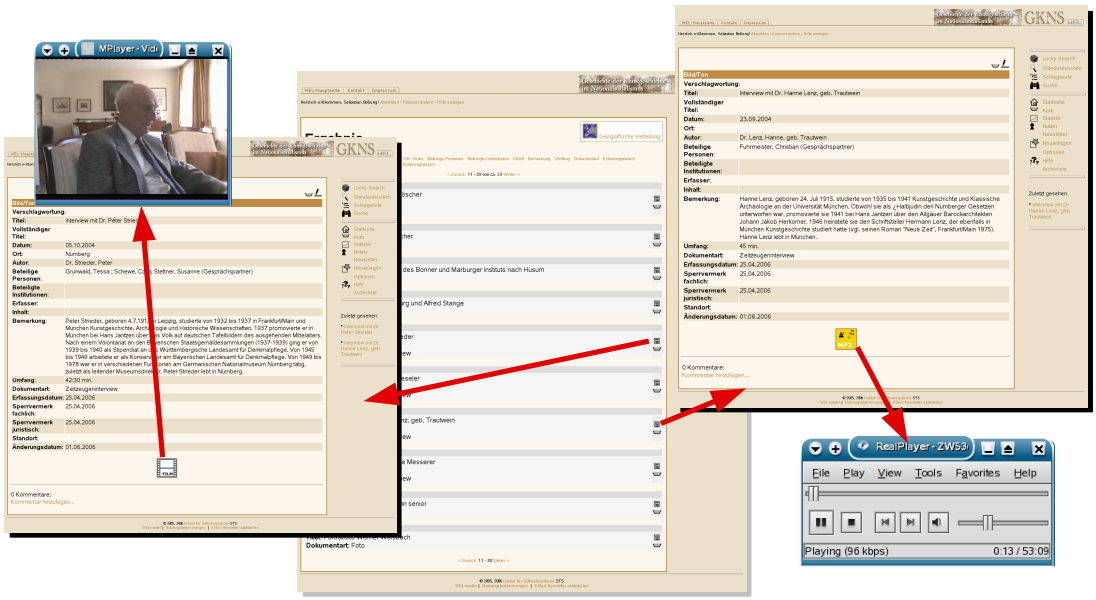


Figure 2.11: *Linking among hypermedia documents in a web-based hypermedia system.*

The term "hypermedia" was first used in 1965 [Nel65] expanding the application of the hypertext approach to different kinds of documents. An overview of hypermedia systems can be found in [Con87]. The most famous example of a hypermedia system that is also in widespread use today is the world-wide web [BLCGP92]. Figure 2.11 shows a collage of screenshots from a web-based hypermedia system.

Hypermedia systems usually allow the linking of any two of their resources. The system cannot control whether such a linking is sensible. Some checks can be made with a more formal model of hypermedia [MGMW05] that helps to capture semantics contained in both the documents and the links between them. Such a formal model can, e.g., be a conceptual model [BVA+97, MMWR01]. Hypermedia documents as well as links between them then become instances of classes from the conceptual model. Constraints can be expressed to allow only certain types of links between specific classes of documents.

However, a strict formal model also has disadvantages [TW86, SM99]. There is a danger of cognitively overloading users as formal models tend to force them to produce more fine-grained descriptions. Fine-grained descriptions also make it more difficult to convey tacit knowledge that expert authors are not fully aware of possessing. Imposing a predefined structure can be problematic as authors are then confined to expressing themselves in this structure without the possibility to start with a broad overview and to later refine their thoughts.

A "scale of formality", which ranges from plain documents to explicit structure with explicit meaning, can be established to evaluate the trade-offs between author and reader formality [MGMW05]. By assessing the degree of formality found in semantics representations for readers, authors, and the system, a comparative analysis can be performed that pinpoints areas in which "semantic gaps" can occur. An example of such a gap is a system that uses a highly formal internal representation which is exposed directly to readers and authors.

Recently the focus of research in hypermedia systems has shifted towards knowledge-based systems with particular regard to the semantic web [BN04, WAS06]. Contemporary hypermedia systems do not only provide associative organization of information, but incorporate sophisticated user models to be employed in presentation and information selection. An important application field is electronic learning.

2.3.3 Semantic Descriptions

This section discusses some approaches to semantically describe multimedial content. The descriptions are made for computers who have little cognitive means to deal with this content—usually none at all. To provide semantic access to content to a computer, it is therefore necessary to (a) build a semantic description of what is perceived in the content by a human, and (b) link the parts of this description to the multimedial parts of the content.

A multifacet formal image model is introduced in [Mec95] . It is constructed for information retrieval purposes. The model distinguishes two views of the image: a *logic* and a *physical* view:

- *Physical view*: A bitmap representation of the image. Several color encodings are available.
- *Logic view*: An integration of all aspects of the image. To create a representation that is as complete as possible, this view is structured into four sub-views:
  1. *Structural*: Defines which objects are of importance in the image. Identified objects can be composed of other objects to obtain a tree describing the objects found in the image.
  2. *Spatial*: The layout of the image is described in a separate spatial view in terms of spatial extent of the image objects, their overlap, etc.
  3. *Perceptive*: The perceptive view provides the basic visual properties of the image objects: color, brightness, and texture.
  4. *Symbolic*: This view associates semantic descriptions with complete images or individual image objects. Several semantic models are considered for description.

All views serve as background information which the information retrieval process is built upon. They are targeted at computers and consequently not shown to human users. Based on the views, weighted models for image-based information retrieval can be built [MCM05], which aim to integrate human perception characteristics into the retrieval process. Many other models for image retrieval were built, a summary of trends can be found in [HTS+06].

To provide descriptions of resources on the world wide web, the Resource Description Framework (RDF, [Bec04]) can be used. It provides the means to make statements about resources that can be identified by a Unique Resource Identifier (URI, [BLFM05]). Basically, RDF statements consist of a subject, a predicate, and an object, e.g., to express authorship of a webpage, one could state:

http://www.sts.tuhh.de/se.bossung has an *author* whose value is *S. Bossung*

Along with other properties of this webpage this is also shown in the RDF graph in figure 2.12 which consists of three statements about the webpage. As shown in the figure, the nodes in an RDF graph can be further URI references or literals. Therefore, RDF statements are essentially triples of URIs in which the URIs are used to refer to what the statement is about. The use of URIs for this purpose restricts the expressiveness of statements as some things are not easily referecable by URI. RDF provides a plain text as well as an XML syntax. Literals can be typed in primitive types, for example those of XML Schema (see section 2.4.1).

The fact that all entities are identified indirectly by reference as is illustrated by the following example (from [Bec04], URIs slightly abbreviated):

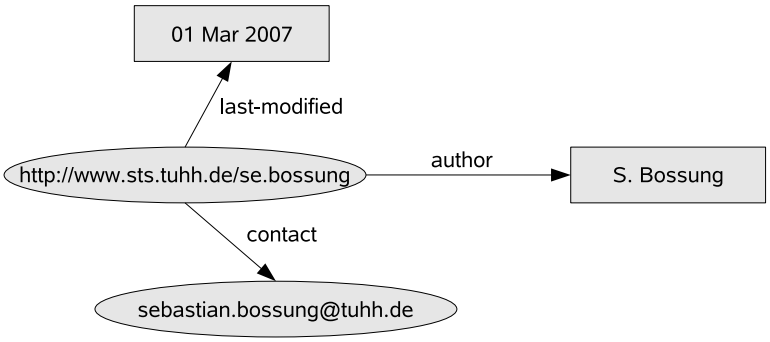


Figure 2.12: *A simple RDF example. URI references are given in ellipses, literals in rectangles.*

```
<rdf:Description rdf:about="http://.../rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage>
      <rdf:Description rdf:about="http://.../net/dajobe/"/>
      </ex:homePage>
    </rdf:Description>
  </ex:editor>
</rdf:Description>
```

This example even omits all necessary namespace declarations for the prefixes `rdf:` and `ex:`. The high amount of URIs in RDF statements makes it difficult for humans to work with RDF directly, requiring tool support [QKH03] for such basic tasks as creating RDF statements.

In plain RDF any resource can be described by any predicate with any further entity. This freedom can make the use of RDF in specific application domain difficult. Users of these domains usually have a clear understanding of the domain and communication between users is done through agreed vocabulary of the domain. To restrict the vocabulary that is available in an RDF description, RDF Schema [RDF04] can be used.

The Web Ontology Language (OWL, [BHH+02]) uses RDF and RDF Schema to express ontologies. Its purpose is to formally describe the vocabulary used in RDF descriptions of web resources. OWL is made to be machine-readable and is not intended for consumption by humans. Rather, OWL documents are authored through tool support which benefits from extended services [LBF+06]. OWL provides three sublanguages with increasing expressiveness at the cost of increasing difficulty of providing services on them.

In the context of the semantic web, a standard has emerged that aims to suit most applications in the field of information retrieval based on multimedial content. The MPEG-7 standard [Mot03] put forth by the Motion Picture Experts Group deals with the description and annotation of multimedial content. A key motivation for the creation of MPEG-7 was the simplicity—perhaps more adequately the poverty—of previously available descriptions of multimedial content. It was "necessary to develop forms of audiovisual information representation that go beyond the simple waveform or sample-based, compression-based (such as MPEG-1 and MPEG-2) or even objects-based (such as MPEG-4) representation" [Int04]. Therefore, MPEG-7 builds on these low level descriptions of the previous MPEG standards and adds additional layers of abstraction. It is expected that this will enable richer multimedial application which can provide more useful capturing and retrieval means to users.

The prime focus of the standard is on audiovisual content, but this is not a limitation in principle. A variety of abstraction levels can be used for description, starting with very low

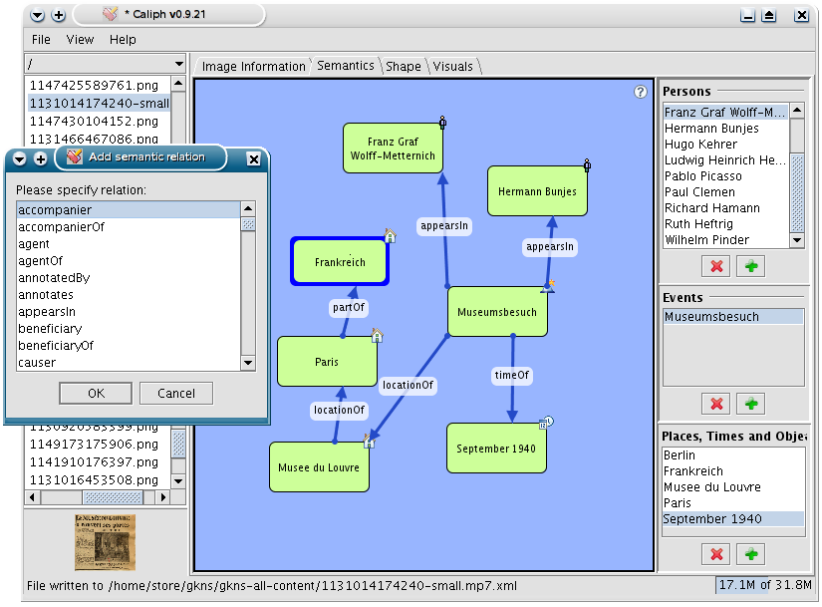


Figure 2.13: *The Caliph tool for capturing MPEG-7 media description on photographs [LBK03]. The semantic description in the middle part captures a portion of the document that is also shown in figure 8.5.*

level descriptions which deal with the primary building blocks of the content, e.g., colors or textures of images. On higher levels of abstraction means are available to sum up these lower level descriptions. Common examples are chord patterns of music or visual areas in images. Descriptions refer to units of content called *features*.

The MPEG-7 standard is divided into several parts including:

- *Multimedia Description Schemes.* The description tools dealing with generic features and multimedia descriptions.
- *Visual and audio description tools.* For the features of the multimedia documents, syntax and semantics of their descriptions (called *descriptors*) are defined. The means to create relationships (in *Description Schemes*) between these features are provided.
- *Description Definition Language.* Used to define the available descriptors to describe multimedial documents as well as for extending the description capabilities by providing new Description Schemes. The Description Definition Language is based on XML Schema but adds some MPEG-7 specific features.
- *Schema Definition.* Specifies the schema using the Description Definition Language.

Other parts of MPEG-7 include systems, software, profiles, testing etc., but these are not of direct interest here. A complete list of the parts of MPEG-7 can be found in [Mar02b] or [Int04].

MPEG-7 thus provides means to describe content at various levels. Typical applications include image/video annotation and query, content classification, but also low level descriptions of audio [KM05]. Figure 2.13 shows an example of an image annotation tool. Users are able to capture technical metadata of the image (similar to those usually stored in EXIF [ASE02]) and manage low level descriptors (such as color distribution), but can also create semantic descriptions of the image. The screenshot in figure 2.13 shows an example where temporal and spacial entities are interrelated. The entities are created and managed separately. Such

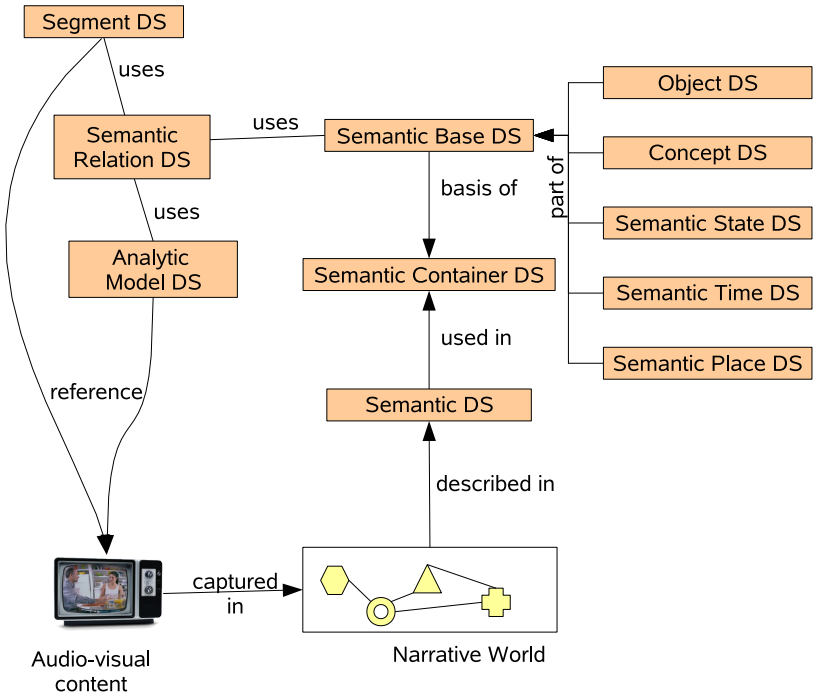


Figure 2.14: *MPEG-7 Semantic Description Schemes [Int04].*

a description provides a structured representation of the image. The advantage is that to a machine the image is no longer one opaque string of bytes and retrieval can be based on the entities depicted in the image. There also is a corresponding retrieval tool, see [LBK03] for more details.

The structure of the semantic descriptions as well as their interrelation with the low level descriptors (Segments and Analytic Model) is depicted in figure 2.14. The central idea is that from an abstract SemanticBase description scheme several concrete schemes are derived, covering time, place, events, etc. On the one hand, a semantic description of the multimedial content is created from descriptors according to these semantic schemes. On the other hand, the content is also annotated with low level descriptors which can be put into relations with the semantic descriptors. This creates an integrated representation of the content.

The MPEG-7 standard provides many means to structure the content and proposes approaches to semantically describe the structures. These means can also be applied to larger collections of content. As the amount of content grows larger, ensuring coherent semantic descriptions becomes difficult. MPEG-7 alone focuses on individual pieces of content and does not cope with semantically interconnecting diverse collections (as is, e.g., necessary for semantic web applications [Tro03]). In essence the problem is that MPEG-7 provides means to attach semantic descriptors to content, but does not provide help in defining the semantics of the descriptors used. Several authors (e.g., [HBHV04, Tro03]) have noticed this semantic integration problem and propose the combination of ontologies with MPEG-7. The aim is to build large collections of diverse content with coherent semantic descriptions. An ontology to represent MPEG-7 metadata terms has been presented in [Hun01]. Based on this ontology, the semantic descriptors for a particular application can be defined [Tro03].

Kosch [Kos02] notes that MPEG-7 lacks some concepts of multimedia databases such as ways to introduce different levels of abstraction for media segmentation. He also considers the semantic model, which MPEG-7 provides to describe narrative worlds, to be rather weak because it is not extensible enough.

By means of semantic descriptions, MPEG-7 enables the creation of descriptive representation of the multimedial document. Its overall meaning, however, is outside the scope of MPEG-7. Along the same line, MPEG-7 does not deal with the interrelation of documents, especially not with those of segments of one document to other documents. The definition of the semantics of the descriptors (even the semantic ones) are left to other technology, such as ontologies, and no attempt is made to cover these semantics through, e.g., description with other multimedia documents.

2.4 Introduction to Some Technologies

This section briefly introduces some technologies that are of interest in the implementation of systems which support work on semantic descriptions of multimedial content. These systems will be presented in chapter 5.

2.4.1 XML Schema

XML Schema is a language to specify the grammar of XML documents. XML Schema definitions are themselves proper XML documents such that they can be edited with the usual XML tools. XML Schema is a rather elaborate W3C recommendation (see [FW04] for an overview), only some of its parts will be described here.

The general approach to XML Schema is to define the structure of each XML element that can occur in documents adherent to the definition. Such definitions are generally referred to as *schema*. The schema defines the structure of elements in basically two ways: by listing their attributes and by their substructure, that is, the child elements that can be contained in the described element. XML Schema assigns a type to each node in the XML document tree. This type can be either *complex* or *simple*. It is complex if it describes child elements, it is simple otherwise. Plain data such as character string, dates, numbers, and booleans are covered by simple types that are built into XML Schema. A complex type could be defined as:

```
<xs:complexType name="TypeA">
  <xs:sequence>
    <xs:choice>
      <xs:element name="this" type="xs:string"/>
      <xs:element name="that" type="xs:string"/>
    </xs:choice>
    <element name="last" type="xs:integer"/>
  </xs:sequence>
</xs:complexType>
```

The elements conforming to type `TypeA` contain two elements: First an `xs:string` which is either called `this` or `that` followed by an `xs:integer` in a `last` element. While the type does define the names of sub-elements it does not define the name of the element(s) that comply to it. This is important to facilitate reuse. Attributes can also be declared in a complex type:

```
<xs:complexType name="TypeA">
  <xs:sequence>
    ...  </xs:sequence>
  <xs:attribute name="attribute1" type="xs:dateTime"/>
</xs:complexType>
```

Attributes are declared last and must be of simple type as an attribute in XML must have a character value. Admissible elements are defined alongside their types:

```
<xs:element name="elementA" type="TypeA"/>
```

Element declarations are also used inside complex types as in the above example of `TypeA`. Of course, the elements declared in complex types can again be of complex type.

XML Schema allows the definition of new types based on existing ones. This is called an *extension*. Extensions can be made from both simple and complex types. With simple types, extensions can, e.g., restrict the range of integers or define a regular expression to be observed by character strings. Complex types can define additional sub-elements and attributes:

```
<xs:complexType name="TypeB">
  <xs:complexContent>
    <xs:extension base="TypeA">
      <xs:sequence>
        <xs:element name="newInB" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:compexType>
```

Extension of complex types does no effect on substitutability of elements. An element of an extended type cannot necessarily be substituted for an element of the base type. Instead, the substitutability of elements is defined explicitly by placing elements that are substitutable for each other in *substiution groups*:

```
<xs:element name="woman" ... substitutionGroup="human"/>
<xs:element name="man" ... substitutionGroup="human"/>
```

This means that anywhere an element from the `human` substitution group is declared, any other element from this group may also occur.

2.4.2 XML Inclusions

The XML Inclusions recommendation ([MO04], abbreviated XInclude) put forth by the W3C specifies the inclusion of XML documents into XML documents. Inclusion facilities are often found in programming and markup languages to enhance modularity. Inclusion facilities for specific purposes have been part of many particular XML grammars. XIncludes defines a general purpose facility to enable generic support of inclusion by XML authoring and processing tools.

XInclude defines a processing model to enable such modularity for XML documents. XInclude does not work at presentation level (such as, e.g., the embedding of a media object in an HTML page) but on the information sets representing the XML documents. This means that it is media type dependent and can only include XML documents into other XML documents.

The two documents that are related by an inclusion, remain independent entities. This means that both documents are loaded and parsed independently. Therefore, a failure to (temporarily) retrieve a referenced document is not necessarily a fatal error.

Inclusions are defined in an XML-friendly manner by an `<xi:include>` element. It is the purpose of this element to specify the document to be included by means of a URL as well as the manner of its inclusion. A simple example of an inclusion thus is:

```
<?xml version='1.0'?>
<my:document
  xmlns:my="http://example2.org"
  xmlns:xi="http://www.w3c.org/2001/XInclude">
  <xi:include href="http://example.org/other.xml"/>
</my:document>
```

In addition to this, XInclude provides meachanisms to include only parts of an XML document or to include the referenced document as plain text. Document authors can also specify fallback content to be used if the referenced resource cannot be accessed.

```
declare namespace bill="http://localhost/testbill/bill.xsd";
declare namespace n="http://localhost/bill_summary";
declare function local:totalValue($bill as element) as xs:double {
  sum(for $l in $bill/bill:line return
    ($l/bill:qty * $l/bill:itemPrice))
};
<n:result>{
  for $day in ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
  return
  <n:day name="{$day}"> {
    for $bill in /bill:bill[contains(bill:due-date, $day)]
    return
      <n:bill-summary>
        <n:short-sender>
          {$bill/bill:addressing/bill:sender/bill:company/text()}
        </n:short-sender>
        <n:short-receiver> {...} </n:short-receiver>
        <n:item-count>{count($bill/bill:line)}</n:item-count>
        <n:total-value>{local:totalValue($bill)}</n:total-value>
      </n:bill-summary>
    }
  </n:day>
} </n:result>
```

Figure 2.15: *Example of a more complex XQuery that uses many of the features described in this section. The query creates a well-formed XML document with bill summaries aggregated by day of week.*

2.4.3 XML Query

The language XML Query (XQuery) is a language to pose queries against XML documents. This section will give a very brief introduction to the parts of XQuery that are used in this thesis. A more complete description can be found either in the W3C recommendation [BCF+05] or in an introduction by Philip Wadler [Wad02]. This overview follows the one given in [Bos04].

XQuery is a functional language that attempts to fulfill the needs of two comunities: Those seeing XML as documents (largely coming from the HTML or plain SGML worlds) and those treating XML as data (mainly in semi-structured databases and data exchange between systems). XQuery uses XPath as a technique for selecting nodes. The return type of any expression in XQuery is a sequence of XML nodes (even though many of these only contain zero or one values). A sequence is ordered and may contain duplicates.

Some important XQuery constructs are described below. Figure 2.15 shows a more complex query that uses these elements.

XQuery allows the *construction* of new nodes (elements, attributes, text, etc.). The syntax for this is the same that would appear in an XML document which contains the respective node. The following code will for example construct an element node called "result" with an attribute called "name" (and value "first result"). The element contains the text "123":

```
<result name="first result">123</result>
```

Enclosed expressions are closely related to constructors. Curly braces "{" and "}" are used to disambiguate expressions nested inside constructors from literal text. Consider this example:

```
<result>{ $a }</result>
```

The contents of the "result" element will be the value of variable `a`. By contrast, without the curly braces the contents of the "result" element would be the text literal "$a".

XQuery uses *path expressions* for selecting nodes. XPath expressions are made up of several steps that start from a sequence of context nodes, e.g., from the root of a document. The different steps are deliminated by forward slashes "/".

```
/bill/sender[@name="Peter"]
```

This will start from the document root and select the element `<bill/>` and then descend to its child `sender`. All `<sender/>` elements that are children of a `<bill/>` element are returned if they have a `name` attribute (denoted by the "@") with value "Peter"'. XPath is not limited to navigating from an element to its children. It allows different navigation paths called axes. Another important axis is "descendent-or-self": the node or any of its descendents. For example

```
/descendent-or-self::*
```

simply selects all nodes in the document. Results of path expressions are—of course— sequences of nodes.

Iterating over a sequence of nodes is important in XQuery. First selecting a sequence of nodes via a path expression and then iterating over it to perform a transformation or computation is common practice. The syntax of the *FLWOR* (for-let-where-order by-return) expression is illustrated in this example:

```
for $bill in /bill
let $name := $bill/sender/@name
where $bill/@total-sum > 100
order by $bill/@total-sum
return
  <sender>{ $name }</sender>
```

The first line is an iteration over all nodes in `/bill`, where in each iteration the current node is bound to the variable `$bill`. The next line then binds an additional variable `$name` to the name of the `sender` of the bill. The `where` clause filters the sequence over which to iterate to only contain bills for which the `total-sum` is over 100. `Order by` sorts the bills by their `total-sum` attribute. Finally `return` gives the expression that is evaluated for every iteration with the new variable bindings.

The output could look like this:

```
<sender>Peter</sender>
<sender>Texaco</sender>
<sender>Real Stationary</sender>
<sender>Infinite Computers Ltd.</sender>
```

Note that this is not a well-formed XML document.

XQuery has many built-in functions but also allows users to specify their own. Calling functions is quite simple:

```
<overall-sum>{ sum(/bill/@total-sum) }</overall-sum>
```

This example sums the totals of all bills and puts the result as the contents of the `overall-sum` element. Note that the `sum` function works on a sequence of values. User defined functions are also possible. As an example consider the function `totalValue` in figure 2.15. The function is declared in two parts: The signature and the body expression. The signature gives the namespace (`local` in this case), the function's name (`totalValue`), the arguments along with their types (`$bill` of type `element`), and the return type (`xs:double`). The body can be any XQuery expression. The example iterates over all `line` elements under the element passed to the function and multiplies the values of their `qty` and `itemPrice` elements. The `sum` function then computes the total of all the values in the sequence, this value is returned by the function.

Chapter 3

Core Asset Expression Language

During the course of our life, each of us acquires large amounts of knowledge about our surroundings. When confronted with a new situation, we make use of this knowledge to interpret the objects and events at hand. Such an interpretation can only be successful if we are equipped with the appropriate contextual knowledge to deal with the new situation.

As an example, consider a photograph of a street scene. Recognizing it as such requires familiarity with various concepts such as cars, pedestrians, shops, or traffic lights. As these are concepts of everyday life, most people of contemporary cultural heritage will have no problem with their identification and their meaning. However, if the photograph were to depict something more particular, not everybody would be able to make the required interpretations.

Medial content—such as a photograph, a painting, a movie, or a written text—can be interpreted and understood in much the same way: by application of the appropriate parts from our contextual knowledge. This contextual knowledge is called the *context* in the following. Obviously, any two persons are likely to have different contexts. This can for example cause them to interpret the same medial content differently. One person might feel threatened by a certain image, while another might not. The contexts can also differ in terms of what is contained in them. This can cause one person to not be able to interpret content because of the lack of knowledge about central parts. Another might be knowledgable about those parts and therefore able to interpret the content. The latter person could then explain the meaning of the content by conveying the meaning of the missing parts.

The Asset Expression Language (AEL) is used to provide such explanations by giving conceptual abstractions for pieces of content. The expressions represent real-world entities in both a medial as well as a conceptual way. Existing expressions can be combined into larger ones. The AEL also allows for substructuring of content to refer to individual pieces. Asset Expressions borrow the concepts of abstraction and application from the λ-calculus.

The description of the meaning of an entity in terms of the meanings of its subcomponents is not uncommon. It is used, for example, in the denotational approach to define the semantics of programming languages. Stoy [Sto77, p 20] notes that "in the denotational definition, [...], the value of the program is defined in terms of the values of its subcomponents" (also see section 2.2.2). This is applied in a divide-and-conquer manner until it ends in subcomponents which are simple enough to be interpreted correctly. What exactly these atoms are depends on the intended audience, who has to be able to interpret these atoms based on their contexts.

A similar notion is used here for medial content: If the whole of the content cannot be understood in one piece, some explanations of its key elements are given. These explanations can again use medial content. Therefore, small-scale explanations are recursively repeated to build larger structures which capture a portion of some domain. It is hoped that—if the gap in contextual knowledge of the audience is not too large—this can eventually end in descriptions which can be understood given the particular contexts of the audience.

This chapter describes Asset Expressions which are used to handle these issues of lacking

contextual information when dealing with medial content. They do so by using medial content to explain parts of another content. More specifically, their workings can also be broken down into several means. The first is the identification of gaps in contextual information, which have to be filled. Next, particular pieces of the content can be identified that require these further explanations because they cannot be understood from contextual information alone. Finally, existing or new explanation structures can be used to fill the gaps in the context in connection with the parts identified in the previous step.

The term "content" in this work refers to an instance of a multimedial resource. Examples include images and audio recordings, but the notion of content adopted here is broad enough to also span plain text or combinations of other formats. Content is in other work sometimes also referred to as "documents", "multimedial artifacts", etc.

3.1 Plain Asset Expressions

The problem of missing contextual information does not only occur when interpreting medial content. When programming a computer, one runs into similar issues. Any sequence of statements might for example make reference to memory locations in order to retrieve parameters to an algorithm. These references are not obvious[1] from the statements themselves. Consider as an example the following pseudo code:

```
i = 1:  integer; j = 1:  integer; k:  integer; n:  integer
for count from 0 to n do
begin
  print i
  k := i + j; i := j; j := k
end
```

It is not apparent at first glance that one needs to bind a value to `n` (but not to `k`) in order to meaningfully invoke the algorithm.

A common notion in computer science is that of signatures. The key idea is to identify and make explicit any unbound variables in the part of a program that is covered by the signature. This provides users with information on what they need to supply before invocation. By defining a function signature, it can be made clear that a value for `n` is required:

```
function fibu(n: integer): integer
begin
  // code from above
  return k
end
```

This is akin to requesting additional contextual information to be able to interpret medial content.

Applying a similar notion to content, one observes that some parts of this content require additional information to allow the content to be understood. These parts (free variables in the programming language analogy) are bound in the signature over the content. The content is considered to *denote* a real-world entity. The content is interpreted by the user, the result of this interpretation is the denoted real-world entity. This interpretation can only be carried out if all variables bound in the signature are supplied with actual parameters. In other words: Users might lack some contextual knowledge, which is needed for a full understanding. Signatures over function bodies point out the free variables used in the function, signatures over content do the same for contextual knowledge. As illustrated in figure 3.1, Asset Expressions capture this denotation by building larger structures.

[1] In many cases they can be derived but for more complex code it is certainly difficult to determine.

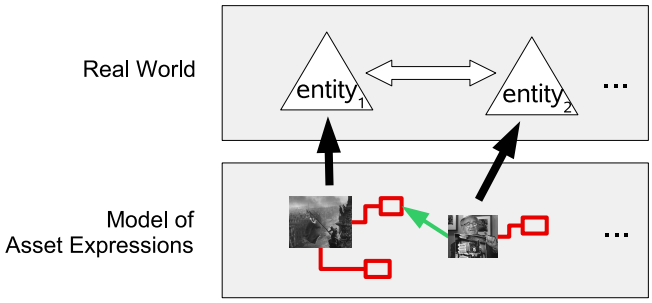


Figure 3.1: *Asset Expressions model entities from the real world. Through abstraction and application different expressions can be related, thus denoting relationships between the respective entities.*

A very concise yet powerful approach to specify signatures is the λ-calculus (see section 2.2.1 and [Rev88]). In its simplest form it is based on only three concepts: variables, abstractions and applications. Asset Expressions borrow the concepts of abstraction and application to express signatures over content and the application of explanations, respectively. The core syntax of Asset Expressions is similar to that of the λ-calculus, it will be detailed in this section.

As Asset Expressions allow for abstractions over any multimedial content, this does introduce the problem that variables in the content cannot be identified by name equality as is common in the λ-calculus. In the λ-calculus, occurrences of variables that are bound through abstraction are easily traceable in the expression by name equality and scoping rules. In multimedial content, other means of pointing out the variable in the content are required (content does not contain the symbol x). This is handled by deep content structuring in section 3.2. Section 3.3 introduces a typed version of Asset Expressions.

3.1.1 Syntax

Asset Expressions must deal with medial content which is to be shown directly in the expression itself. Even if—for purposes of abstraction and application—a name for the medial content were used instead of the content itself, this name had to be bound somewhere, ultimately making it necessary to show the content verbatim. Besides this, it is felt that direct abstraction over the medial content enhances the user's ability to interpret the expression as the entity it denotes. If this is desired, Asset Expressions offer means to bind a particular content to a name. Therefore, the syntax of Asset Expressions is not a plain textual one but includes representations of medial content as well.

A simple example which explains the person shown in a painting is:

(λperson. )Napoleon

The syntactic elements are described below.

Content

Any multimedial content can in principle be used in Asset Expressions. More precisely, a representation of the content can be used. The choice of content types which are available to

construct Asset Expressions might be limited by the medium used to present the expressions. In some cases a preview of the content may be more feasible for particular types of content (e.g., video or audio when the expression is presented on paper). This preview or the representation of the content is shown directly in the expression. In the plain form of Asset Expressions, content remains an opaque whole which has no further substructure. The potential of substructuring is explored in section 3.2.

Naming Expressions

For further reference expressions can be bound to a name. This name can be used in other expressions to refer to the named expression. The syntax for binding an expression to a name is:

name := *expression*

The name can then be used in place of the expression. A name definition evaluates to the expression the name is bound to.

Abstraction

Abstractions are shown in style of the λ-calculus:

λvariable.*expression*

Abstraction is right-associative. The above means that an explanation is required for this expression to be understood. This requirement is given a name (variable) for future reference. As in the λ-calculus, the abstraction creates a function with one parameter, see section 2.2.1. One can for example require an explanation of a person shown in a painting:

λperson. 

There may, however, be several explanation requirements in the content that is abstracted from. Multiple abstractions are then used to capture these:

λinscription.λperson. 

To express such multiple-argument signatures, the notion of *currying*[2], known from functional languages in general and the λ-calculus in particular, is used here as well. The idea of currying is to transform a function that takes several arguments and returns a result into a function with one argument. The function with one argument returns as its result a new function that requires the remaining arguments of the original function. The signature in the second example includes two parameters: inscription and person. By means of currying the first abstraction constructs a function with one parameter (person), the second abstraction is performed over this function introducing the second parameter inscription.

[2]The notion of currying was named by Christopher Strachey after the logician Haskell Curry. Currying was introduced by Schönfinkel [Sch24] and extensively used by Curry.

Application

Explanation needs identified by abstractions can be met by means of applications. A second expression is applied to an abstraction. Asset Expressions follow the syntax of Revesz [Rev88] as the use of whitespace in applications can make expressions difficult to read in the presence of multimedial content.

$(expression_1)expression_2$

$expression_1$ is applied to $expression_2$. Application is right-associative. Usually, there will be an abstraction in $expression_1$ to match the application, but this is not enforced by plain Asset Expressions. The expression $expression_1$ is also called the operator, $expression_2$ the operand. An application which corresponds to an abstraction is said to *meet* it. Vice versa the abstraction is called *filled*.

Consider the following example:

(λperson. )Napoleon

It is assumed that the expression Napoleon has been defined previously. Several abstractions are matched with several applications:

((λinscription.λperson. )References)Napoleon

A pair of abstraction and application is also called an *explanation* as it is normally created to provide information on (i.e., to *explain*) some aspect of the content that is abstracted from. In expressions without open abstractions the order in which abstractions and applications are given does not matter, as long as they match. The previous expression denotes exactly the same entity as:

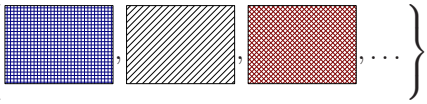
(λinscription.(λperson. )Napoleon)References

Lists

Lists of Asset Expressions are enclosed in { and }. The elements are separated by commas:

$\{e_1, \ldots, e_n\}$

Lists are valid Asset Expressions and can be used anywhere a simple expression is expected. This is for example useful to deal with content which comes in multiple parts such as scanned pages of a document:

{  , ...}

As they are normal Asset Expressions, abstractions and applications can be used over lists, or lists can be used as the operand of an application:

$(\lambda x.\{e_1, \ldots, e_n\})\{f_1, \ldots, f_m\}$

In this example, the list of expressions $\{f_1, \ldots, f_m\}$ describes some joint aspect of the

list of expressions $\{e_1, \ldots, e_n\}$. Note that the abstraction pertains to the whole of the list $\{e_1, \ldots, e_n\}$. The operand can again be a list (as in this example) but does not have to be: there is no matching between the elements of the lists.

A simple grammar of Asset Expressions is:

$expression ::= variable \mid content \mid application \mid abstraction$

$application ::= (expression)expression$

$abstraction ::= \lambda variable.expression$

The full grammar of Asset Expressions is given in section 4.7 after all language elements have been introduced in chapters 3 and 4. This grammar is unambiguous, therefore the abstract syntax tree of expressions is always unique.

3.1.2 Visual Notation

As Asset Expressions deal with multimedial content, a visual notation sometimes suggests itself. After all, a part of the expression is not representable in a textual form anyhow. A visual notation for Asset Expressions is introduced here. It resembles conceptual graphs [Sow00] in its directed connections between entities, however, the notation is not the same. Unlike conceptual graphs, it emphasizes medial representations of entities. The notation is illustrational in nature and will not be used in places where some degree of formality is required. The core primitives of this notation are defined along the syntactical elements of Asset Expressions:

Content

The notation for content remains as it is, the content is shown directly:

Abstraction

Abstractions are shown as boxes in which the name of the variable is given. The boxes are connected to the expression they abstract from, which can be a content.

Note that abstractions are not ordered as in the proper textual syntax because the visual notation does not use currying but allows for several parameters in the signature.

Named Expressions

The definition of named expressions is shown just as in the textual syntax. To refer to a name elsewhere, an ellipse is used:

Application

Arrows are used for applications. The arrow points from the operand to the operator.

There is no special representation for lists, they are lists of visually represented expressions. When applying a list to an abstraction, an allowed shortcut is to show several individual applications instead of the list. This can be used to improve the layout of expressions.

3.1.3 Lifecycle

Asset Expressions are created with the constructors introduced in section 3.1.1. New expressions can be based on existing expressions by referencing these expressions by name:

$e_1 := \lambda x.C$ $\quad$ $e_2 := (e_1)D = (\lambda x.C)D$

The use of e_1 in e_2 has reference semantics. Any referenced name must be previously bound.

$e_1 := \lambda a.e_1$;error, cannot use e_1 on right-hand side, not bound

In general, names can be rebound by the same means used for initial binding:

$e_1 := (C)E$

This implicitly changes e_2 which references e_1 to $e_2 = ((C)E)D$. With plain Asset Expressions, care should be taken with such rebindings as they can heavily change the semantics of referencing expressions. The typed Asset Expressions introduced in section 3.3 take some precautions to limit rebinding issues.

Finally, names can be unbound:

remove e_2

This is, however, only legal if the name is not referenced from any expression. Thus, with the definitions above, e_1 cannot be unbound. remove e frees the name e and deletes the expression bound to it. However, it does not cascade to expressions referenced from e.

3.2 Content Components

In programming languages, parameters of function signatures, such as the one shown in the example in section 3.1, refer to a particular part of the function body, specifically to variables. Variables can be clearly identified in the language used for the body. They can also be connected to the parameters in the signature, the usual means to this end is name equality. Name equality of parameter and body variables is referred to as *Barendregt convention* ([Bar85], figure 3.2(b)) in the context of the λ-calculus. There also exist other means such as de Bruijn's numbering scheme (see [Bru72] and figure 3.2(c)) for free variables. Instead of writing variables in the body as a name, they are simply numbered in de Bruijn's schema and abstractions do not need to mention any name at all. Rather, the connection between abstraction and variable is made by the numbering and the order of the abstractions. Stoy [Sto77] uses visual notation to connect abstraction and abstracted variable, see figure 3.2(d). This notation does not provide a textual syntax for variables at all. Instead, it simply connects the abstraction with the place in the body that is abstracted from.

```
function add(x, y)
  return x + y
end
```

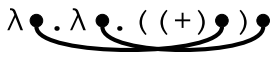
(a) Function in pseudo-code

```
λy.λx.((+)y)x
```

(b) Barendregt

```
λ.λ.((+)1)0
```

(c) de Bruijn

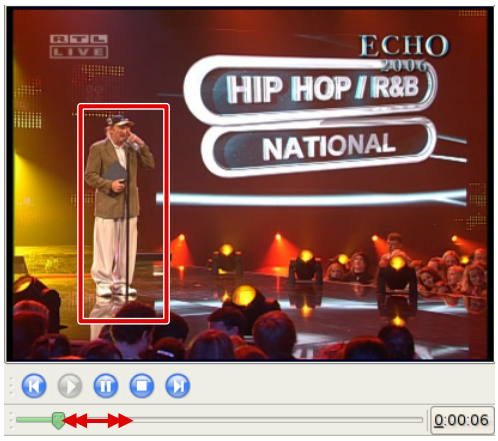


(d) Stoy

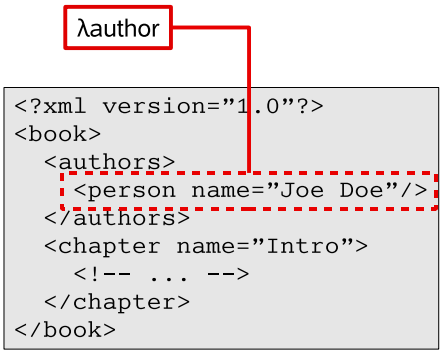
Figure 3.2: *Different ways to connect parameters from the function's signature to variables in the body. Barendregt [Bar85] uses name equality, de Bruijn [Bru72] numbers free variables of expressions, and Stoy uses a visual notation [Sto77].*

Addressing parts of medial content is more difficult than just mentioning a name or doing abstractions in the right order. Different kinds of content are represented in a diverse variety of languages (e.g., bitmap images in pixels, XML documents in text). These languages generally do not introduce the concept of a variable. Unfortunately, this concept can also not be added to them without breaking the content by violating its customary format. Consider the example in figure 3.3(a) and imagine that it is necessary to further explain the person on stage. Doing so first requires this person to be identifiable. In video this can for example be done by specifying the spatial extent of interest and giving a time interval over which it is valid. Abstractions in Asset Expressions make the need for such an explanation explicit. However, in plain Asset Expressions abstractions are made over the content as a whole. There is no mention of any particular part of the content that the need for explanations might arise from. To allow abstractions to be more specific, content can be divided into components. Abstractions are then made over these components instead of over the content as a whole.

Figure 3.3(b) gives an example of a visual notation of such a component and an accompanying abstraction. First, the component is created (dashed box). The subsequent abstraction is over this component. Clearly, one cannot draw dashed boxes into XML documents without conflicting with their nature as XML documents, which includes being plain-text. Means for



(a) Video



(b) XML

Figure 3.3: *Selecting components of a video by combination of time interval and geometric area.*

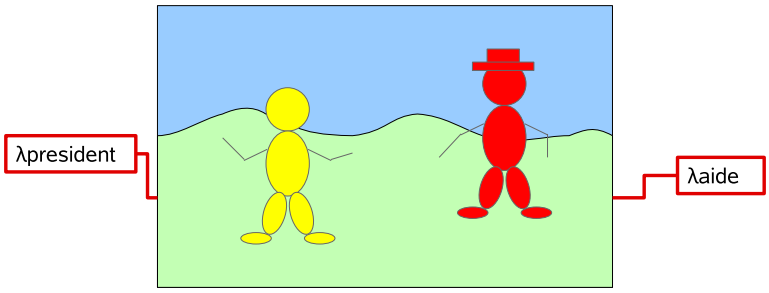


Figure 3.4: *Image with two persons in different roles.*

introducing components into various kinds of content will therefore be discussed below. In addition, there are provisions to handle components and connect them to abstractions.

3.2.1 Components

If an abstraction is created to request an explanation, the creator of the abstraction is often able to point out which particular part of the content is not understood. This also allows the subsequent application to be more precise. In the context of several abstractions over a single content ambiguities might arise without the connection of abstractions to a part of the content. Consider as example the picture in figure 3.4 which shows multiple persons in different roles. To explain the concepts of president and aide, corresponding abstractions are created to be filled with applications of expressions giving details on presidents and aides. However, if the abstractions over the photograph cannot make a connection to the medial presentation of the person they refer to, much is lost. Clearly, it can be important to the understanding of the picture to know who is the president and who is the aide.

To allow this kind of detailed connection between abstractions and parts of the content, content *components* are used. Components are parts of some larger piece of content. Components have a name, which is used to refer to them. The component also needs means of addressing which part of the original content it covers. These addressing means depend on the kind of content at hand, e.g., addressing in images is quite different from addressing in a piece of audio. These addressings are called *selectors* (as they select some part of the content to create the component). Different selectors for various kinds of content will be discussed subsequently. The model for annotations presented in [BCM99] introduces the notion of characteristic structures which are similar to the content components of Asset Expressions in that the base document is composed of such structures and these structures can be subdivided into structures of finer granularity.

Selectors are tuples of addressing information which could be encoded just like records (see [Rev88]) in the λ-calculus. However, considering that the resulting expressions are quite lengthy and that selectors occur rather frequently, an abbreviated syntax is introduced. A selector is simply written as a comma-separated list of addressing information enclosed in pointed brackets:

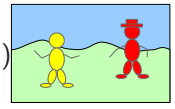
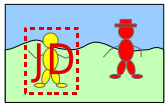
$$<\ldots,\ldots>$$

The part of the content that is addressed by the selector must again be a well-formed instance of this kind of content. This restriction enables composability of content pieces, which is important to language features introduced in chapter 4. For example, a selector over a piece of audio must address a subpart of this audio which can again be played as audio. If addressing were on byte-level on a compressed audio file, this would probably not be the case for all byte intervals. Selectors over XML documents must always result in well-formed XML.

Components are created on content by the application of the function child. This is a built-in function which takes three parameters: (1) the name n of the new component, (2) the selector s to address a part of the content and (3) the expression C in which the component is created.

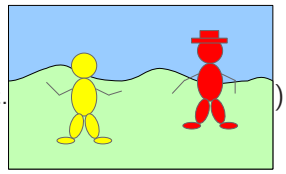
$$(((\mathsf{child})n)s)C$$

It returns a new Asset Expression, e.g., a content, which has an additional component named n identified by the selector s. The name of a component must be unique within the content. With a visual notation to show components of content, this can be used as follows:

e_{comp} := (((child)"JD")<coords-rel, (0.1,0.2), (0.4,0.9)>)  = 

Several components in a single piece of content can be created by repeated invocations of child. With components in the content, there needs to be a way to refer to them by tieing them to abstractions which provide explanations for this part of the content. Without components, one might have written an expression as follows:

PresidentOfAtlantis := ...

(λpresident. )PresidentOfAtlantis

In this expression it is unclear whether the president is wearing a hat or not. To mark the president, a component is introduced into the content as in expression e_{comp}. A direct abstraction over this expression (i.e., the content including a component) could still not achieve the desired effect as there is now an abstraction and a component, but no connection between the two (one needs to keep in mind that there might be multiple components). To make this connection, the right component needs to be exposed for the abstraction. This is done using expose:

$$((\mathsf{expose})n)e$$

Where n is the name of the component to expose and e is an expression in which this component can be found. e does not have to be a plain content, but can be any expression which contains such a content. If a component named n cannot be found in the expression e, this is an error.

Using expose can be thought of as making the named component the currently active one, on which further expressions can be built. For more details see section 4.3, where additional ways to handle components are also discussed. An example is an abstraction over this component:

$\mathsf{e_{pres}}$:= (λpresident.((expose)"JD")$\mathsf{e_{comp}}$)PresidentOfAtlantis

$\mathsf{e_{pres}}$ "pulls out" the component and makes it available to the subsequent abstraction. The abstraction is now specifically over the component and does not apply to the whole of the content. Abstractions over the complete content are of course still possible and also useful. Such an abstraction will for example be created if there is no distinct part of the content representing the particular aspect of the described entity. This is true, e.g., for the artist who created a painting or the epoch it belongs to.

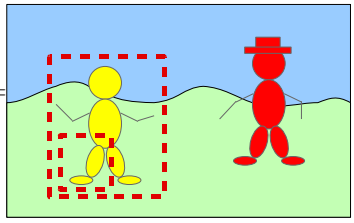
Since they depend on each other, expose will be used about as often as a selector. Thus it makes sense to introduce an abbreviated syntax for exposing components:

$((\mathsf{expose})n)e \equiv [n]e$

The prefix notation is necessary because there might be several exposures for different abstractions from expressions e. The expression $\mathsf{e}_{\mathsf{pres}}$ can now be written more concisely as:

$\mathsf{e}_{\mathsf{pres}} := (\lambda\mathsf{president}.[\text{"JD"}]\mathsf{e}_{\mathsf{comp}})\mathsf{PresidentOfAtlantis}$

Besides abstractions over components, exposure can also be used to define nested components. This is achieved by a combination of child and expose. The selector of a second invocation of child then selects from the part of the content that is addressed by the first selector. In an image, it is for example relative to the area of the outer component:

$(((\mathsf{child})\text{"foot"})\text{<coord-rel, (0.1,0.5), (0.5,0.9)>})[\text{"JD"}]\mathsf{e}_{comp} =$ 

The previous expression is based on e_{comp}, which contains a component called "JD". This component is exposed using the abbreviated syntax ["JD"]. A new component is identified by relative coordinates and named "foot". As the component is created in an expression which has an exposed component, the new component becomes the child of the exposed component. Making the binding of an explanation to a component a two-step process with child and expose has the benefit that further handlings of components besides expose can be offered. These are useful in content construction and will be introduced in section 4.3.

3.2.2 Selectors in Different Kinds of Content

A variety of types of content is available upon which Asset Expressions can be based. These *content kinds* differ fundamentally in their nature. Figure 3.5 shows a partial taxonomy of content kinds. This section introduces some selectors for each of these content kinds.

Inside the pointed brackets, selectors contain a list of addressing information appropriate to the content at hand. To disambiguate different kinds of addressing for the same kind of content, the first element of this list is the selector kind. It states which addressing mechanism is to be used. Thus, the syntax for selectors is:

<*selector-kind*, *address-info*>

The available ways to address a piece of content in a selector are dictated by the kind of content that is selected from. Two dimensional images lend themselves to coordinate-based addressing schemes (measured either in absolute pixels or in relative terms of the image dimensions). Audio content can for example be addressed with time intervals. Content with inherent substructure (such as semi-structured documents) can call for still different addressing schemes.

As a large body of work is available on addressing substructures in multimedial content (e.g., [KM05, LÖSO97, NSS01]), this section does not aim to provide a complete set of selectors for all possible kinds of content. For prominent kinds of content some selectors will be discussed below. Additional selectors for Asset Expressions can be defined by the addressing means provided in the cited literature or those given in the MPEG-7 standard [Mot03] and its extensions. Table 3.1 shows some examples.

First of all, the addressing of content pieces depends on content structure:

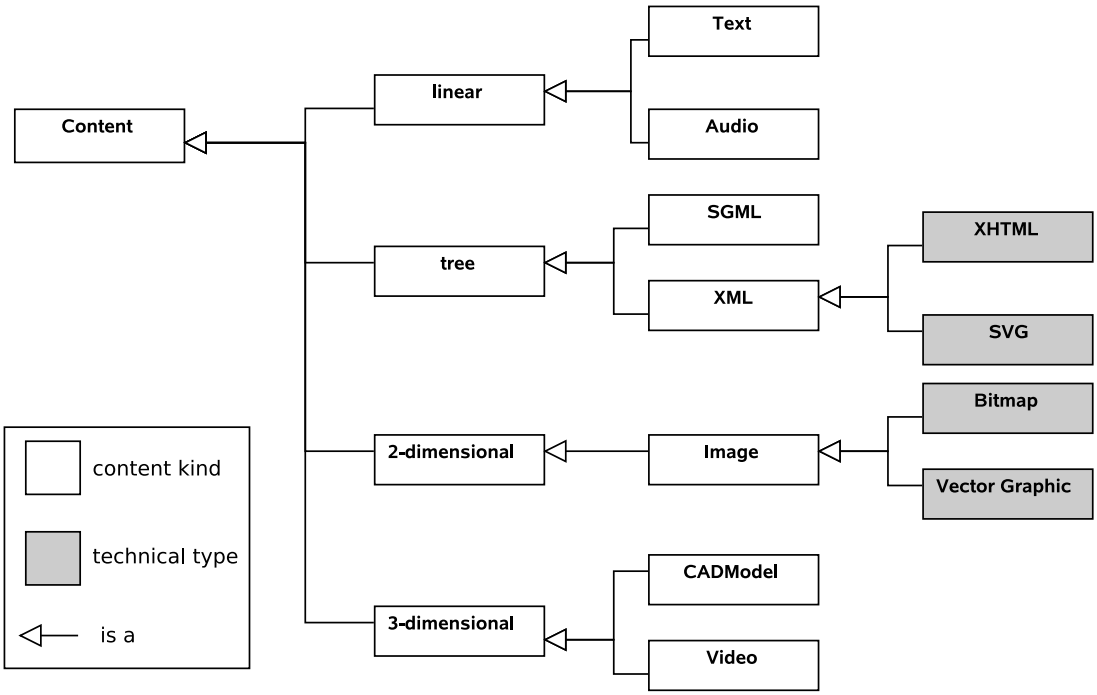


Figure 3.5: *Content kinds with technical content types that are examples for each kind.*

- *Linear.* Examples of linear content include plain text or audio, which are perceived along a single dimension. This dimension can be in time or space.
- *Trees.* This class contains semi-structured document formats, e.g. XML[3], XHTML[4], and other tree-like document formats, but also content with recursive structure such as some vector graphics formats.
- *Two-dimensional.* Representatives of this class are largely images.
- *Three-dimensional.* Includes videos and 3D models. This class often causes some presentational difficulties as current systems mostly use two-dimensional displays.

Different types of addressing can be identified which are largely orthogonal to the content dimensionality:

- *Relative* or *absolute* intervals of the dimensions of the content (e.g., for size of images or the length audio/video) with respect to some defined origin depending on the content.
- *Content-based* or *structure-based* addressing of content parts. The former uses the content itself to address a part, the latter works on the structure of the content without taking the characteristics of a particular instance into account. An example for content-based addressing is the selection of all areas of a certain color/texture in an image. Selection of nodes from a semi-structured document can rely on structure by selecting the n^{th} node on the k^{th} level.

[3]XML: The eXtensible Markup Language [BPSM+06] is a family of languages that share a similar syntax. This facilitates the creation of parsers and other tools to deal with documents in the language.
[4]XHTML: eXtensible HyperText Markup Language, also a W3C standard [W3C02]. XHTML is an XML language.

Selector	Type	Description
<time, 15s, 90s>	3-dimensional, absolute, structure-based	Selector on a video addressing the piece from second 15 to second 90 (could also be used for other time-dependent content)
<xpath, "/book/authors[@id='p15-8'] /text()">	tree, relative & absolute, structure & content	Selector over XML content using an XPath expression which addresses a particular document node from the content
<coords-rel, (0, 0), (0.5, 0.5)>	2-dimensional, relative, structure-based	Selects the upper left quadrant of an image by relative addressing
<xpath, "//img[@src='group.gif']> <coords-abs, (143,201), (255,432)>	chained selector	First selector retrieves an image from an XHTML document, the second addresses a part of this image in absolute coordinates.

Table 3.1: *Examples of selectors*

These types of addressings can often be combined as shown in the example of the XML selector in table 3.1.

The expression e_{comp} in section 3.2.1 uses a selector which is 2-dimensional, relative, and structure-based: <coords-rel, (0.1,0.2), (0.3,0.9)>. coords-rel specifies that relative coordinates are being used to address the piece of content. By the kind of content which the component is created in (two-dimensional image), it is understood that top-left and lower-right corner are given.

Table 3.1 shows some further examples of selectors. The first (<time, 15s, 90s>) can be used for time-based content, such as various kinds of audio or video. It simply gives a time interval. The second (<xpath, "/book/authors[@id='p15-8']/text()">) is meant for use with XML documents as content. The selector first indicates that the addressing information will be given as an XPath[5] selection expression. The subsequent expression then selects a node from the XML document (presumably a string giving the name of a certain author of the book) by making some assumptions about the structure of this document. A complete expression illustrating the use of this selector is given in figure 3.6 as well as in figure 3.3(b) in visual notation.

The representation might show the "actual" structure (e.g., the source of an XML document) or some—possible more user friendly—rendering (e.g., an interpreted HTML document). Addressing can—in principle—be done on both the actual structure and the visualized rendering. An extreme case of visualization-based addressing is the use of pixel coordinates to address parts of (rendered) HTML pages. However, addressing has to take into account potential transformations of the address.

Plain Text

Structurally, plain text does not offer many addressing possibilities. The simplest form is character intervals. However, intervals can also be formed at higher levels for words or even sentences. Plain text can additionally contain line breaks, which can also be used for selectors.

[5] XPath [W3C99] is a navigation language for XML documents. It supports the selection of any node in the document tree.

λauthor:Writer.[a1](((child)
”a1”)<xpath, “/book/authors/person[0]”>)

```
<?xml version=”1.0”?>
<book>
  <authors>
    <person name=”Joe Doe”/>
  </authors>
  <chapter name=”Intro”>
    <!-- ... -->
  </chapter>
</book>
```

Figure 3.6: *Abstraction over a component in XML content.*

Kind	Addressing information	Comment
interval-character	two character counts *begin* and *end*	Selects all characters from *begin* to *end* (inclusive)
interval-word	two word counts	Selects whole words (inclusive), words separated by whitespace
interval-sentence	two sentence counts	Selects whole sentences (inclusive), sentences separatred by full stops
line-character	two pairs of (line number, character count)	Selects from the character given with the first line to the character given with the second line
line-word	two pairs (line number, word count)	Selects from the word given with the first line to the word given with the second line

Table 3.2: *Selector kinds for plain texts. A selector kind* line-sentence *does not seem relevant because there would be two competing addresses of similar granularity as sentences usually span several lines.*

Table 3.2 shows some selectors for plain text based on these addressing modes. Additionally, one might define selectors based on information retrieval technologies to select text based on its meaning rather than on its position.

Thus a selector for plain text might be

<line-word, (10, 3), (15, 8)>

which selects all text from word 3 in line 10 to word 8 in line 15.

Structured Documents

Contrary to plain text, structured documents have a much richer substructure (hence their name). This structure can be exploited to select parts of the document. Generally speaking, most kinds of structured documents are accompanied by at least one query language (e.g., XPath for XML documents or extended O_2SQL [CR94] for SGML documents). These query languages can be used as addressing information in the selectors for the respective documents. Figure 3.6 gives an example of this approach applied to an XML document for which a selector is defined using XPath.

Other means of addressing are also possible. For HTML a combination of URL, document hash code, and XPath expression can be computed and used to identify elements [NSS01].

Kind	Addressing information	Applicable to
coords-rel	two points with coordinates from [0..1]	All images, address relative to total image dimensions
coords-abs	two points with absolute coordinates (i.e., values > 1)	Images which can uniformly be decomposed into discret parts along each dimensions: bitmaps

Table 3.3: *Selector kinds for images. Not all selectors are applicable to all kinds of images.*

Kind	Addressing information	Applicable to
time-sec	a time interval [*start*, *finish*]	all audio
speech-text	a string of text to be found in the speech	spoken content or lyrics
music-timbre	an instrument	music
music-meter	the meter	music with discernable meter
music-key	the key	harmonic music
music-melodycontour	a series of pitches	music with distinguishable single melody

Table 3.4: *Selector kinds for audio.*

HTML might also be addressed based on a document object model, as is in practical use in the Javascript language.

Images

The term "image" is used here to refer to multimedial content which is represented graphically in two spatial dimensions. This includes the usual notions of image such as photographs, illustrations, or graphs. A photograph of a page of text would also be treated as an image. Based on the two dimensions, parts in all images can be addressed by giving intervals for each dimension. The first selector of type coords-rel in table 3.3 uses this.

Additionally, an image might offer further possibilities depending on its type:

- Bitmaps can be addressed in absolute coordinates as they decompose into discreet samples (pixels) along each dimension.
- Vector images are composed of discreet objects, but these are not distributed uniformly along the dimensions. The objects are also more complicated than pixels as they can be of several types and have attributes besides their color.

  An example of a vector-based image format is SVG[6], which is in fact an XML dialect. Therefore, the same selectors as for XML can be used.

  Other vector formats are not necessarily tree-based and therefore require other selectors.

Audio

Using MPEG-7, audio content can be captured at different levels of abstraction [KM05]. First of all, *low-level descriptors* (LLDs) are applicable to all types of audio content. They describe the audio in the time or frequency domains. In principle, intervals from both can be used to define content components. However, it is usually rather difficult to attach semantics to the part of audio that is described in a certain subset of the frequency spectrum. Therefore, of the

[6]SVG: Scalable Vector Graphics [W3C05a] is a description format for two-dimensional graphics composed of complex entities such as circles, lines, and boxes.

Feature	Video Segment	Still Region	Moving Region	Audio Sgmt.
Time	x		x	x
Shape		x	x	
Color	x	x	x	
Texture		x		
Motion	x		x	
Camera Motion	x			
Audio features			x	x

Table 3.5: *Spatio-temporal description features in MPEG-7 for addressing audio-visual content [Mar02a].*

LLDs offered in MPEG-7 the ones in the time domain are most useful for the present purposes as any audio described by using them can accommodate components with time-based selectors. Regardless of its technical format, all audio must be transformed to the time domain before it can be played. Time-based selectors are therefore applicable to all audio. Table 3.4 lists selectors on audio.

On top of the LLDs, MPEG-7 provides specialized descriptors for spoken content and for music. Both have particular characteristics that can be exploited for a higher-level description and also for component creation. The descriptors for speech are based on a series of phonems, i.e., the "atomic" sounds found in spoken language. Groups of these phonems are arranged into words which describe the spoken content. Both phonems and words come from a lexicon. Words are more interesting for content components than phonems because the expression creators are familiar with them and it is also easier to attach meaning to them.

Music can be described by means of some typical properties. Music is played on a variety of instruments each of which has its own characteristic *timbre*. MPEG-7 describes the timbres in quantifyable terms, but more intuitive summaries that correspond to real-world instruments are available [Int01]. Melodies are described with the usual musical terms of meter and key along with a symbolic representation of the approximate melody itself. MPEG-7 refers to the latter as *melody contour*. A precise representation of the melody in MPEG-7 is made as a *melody sequence* which is a sequence of actual notes.

Video

Components in video content can be addressed with existing selectors. Just as in still images, one can, for example, select a spatial region from a video. However, this region usually does not have uniform semantics over the whole playing time of the video. Therefore, video selectors should also be time-dependent. This approach is taken in MPEG-7 which defines three-dimensional segments in video that move over time. An illustration can be found in the usual soccer example [Mar02a]. To distinguish still and moving regions, MPEG-7 provides two models for motion: camera and object motion. Table 3.5 lists some of the spatio-temporal features available in MPEG-7 (left column) and their applicability of different kinds of segments in video.

Heterogenous Content

Individual selectors are defined for a single kind of content. However, heterogenous content, i.e., content which is composed of different kinds of content, is rather common in real-world systems, examples include HTML or PDF[7] documents as well as the multimedia synchroniza-

[7]PDF: Portable Document Format [Ado04]. PDF combines graphics and text into a single file to enhance portability.

(a) Example of heterogenous content [Bra95]

<xpath,"//img[@src='group.gif']> <coords-abs, (143,201), (255,432)>

(b) Selector

Figure 3.7: *Parts of heterogenous content can be selected by chaining multiple selectors. The first selector addresses the image as a whole, the second a part of the image.*

tion language SMIL[8]. To allow the use of heterogenous content, selectors can be chained by giving a number of selectors where a selector is expected, e.g. (for some heterogenous content C):

$$(((\mathsf{child})n)\langle \ldots, \ldots \rangle\langle \ldots, \ldots \rangle)C$$

The first selector is interpreted with respect to the whole content, each subsequent selector with respect to the part addressed by the previous one. Note that the above invocation of child still creates only one component.

As an example, consider an HTML page which contains a bitmap image, such as the one shown in figure 3.7(a). The creator of an Asset Expression for the HTML document would like to provide an abstraction for a part of this image. The appropriate selector can then be composed of two primitive ones: First, a selector for HTML documents is used (employing XPath to select the image as a whole). This selector is then chained with a selector for images, selecting the region from the image as usual (e.g., by absolute coordinates):

<xpath, "//img[@src='group.gif']"><coords-abs, (143,201), (255,432)>

3.3 Typed Asset Expressions

This section introduces typing to Asset Expressions in a type system called $\mathrm{AE}_{\rightarrow}$. The types used are *semantic types* that are motivated by the application domain. They are not defined intensionally, i.e., have no defined substructure. Semantic types can either be simple types, which are embedded in a type hierarchy, a list type derived from a simple type, or function types, which are constructed from two semantic types by means of the $\rightarrow$ type constructor.

[8]SMIL: Synchronized Multimedia Integration Language [W3C05b]. SMIL can be used to combine and synchronize a wide variety of multimedial contents into a single presentation which is combined at the player. Generally, this final presentation could simply be considered to be a two-dimensional, time-dependent video. However, this would discard much structuring information which is already available in the SMIL definitions.

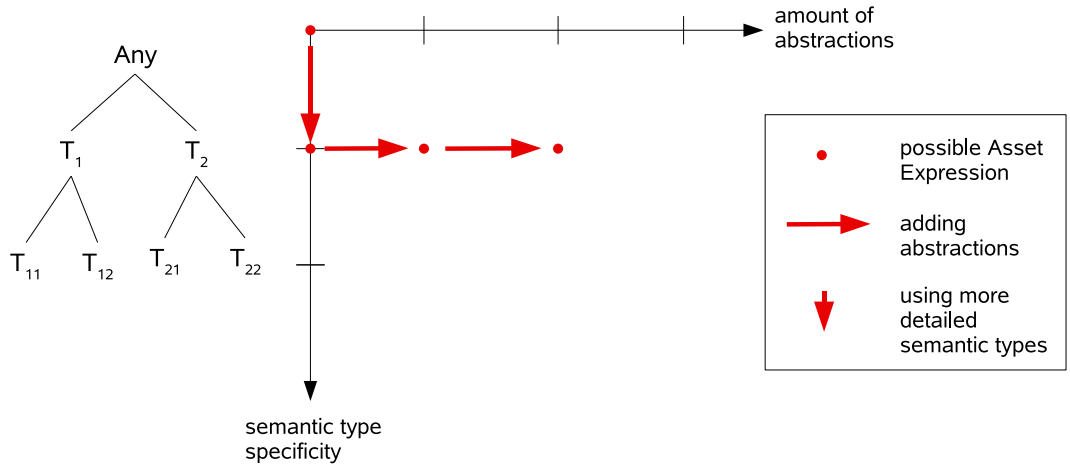


Figure 3.8: *Asset Expressions can vary along two dimensions: The amount of abstractions (i.e., the level of detail of the explanations) and the specificity of the overall semantic type. Both dimensions are independent: An expression of very general semantic type can still exhibit a high level of explanations and vice versa.*

3.3.1 Semantic Types

Content represents a real-world entity in Asset Expressions. The content is interpreted by users and the result of the interpretation is this entity. Semantic types serve to capture the type of the entity that is described by an expression. Each semantic type has a name which is composed of a namespace and a local name that is valid within this namespace. The namespace is part of the name to separate different application domains. For brevity, it is assumed that there always is a default namespace corresponding to the current application domain.

It should be noted that semantic types are not technical in nature, e.g., "JPEG-Image" will generally not be a semantic type. Rather, the type of a content is the type of what it represents. As an example, an image of a student could be typed with the semantic type Person. The technical type is motivated by the content *representation* and is of concern to system implementations only. The technical types of content are also of interest when transforming Asset Expressions for use in other systems, see section 6.2.

Content and abstraction variables are assigned types by the user. The same content can have varying semantic type if used in several Asset Expressions. This mirrors the freedom of the users to explain the same content in different ways to form expressions which denote separate entities. This aspect of semantic types is portrayed in more detail in section 4.6.1.

In Asset Expressions all literals, e.g., 1967 as a year or "Peter" as a string, are considered to be content just like images or videos. Consequently, these "literals" are typed in semantic types, which are assigned by users according to their meaning. The same representation, e.g., 1967 can be interpreted as a Year or a NaturalNumber depending on the context it is seen in.

Semantic types are structured in a taxonomy of types which is defined by the user. Each semantic type—except for the built-in one Any—has exactly one super-type. The restriction to a single super-type is deliberate though not necessary for any technical reason. Rather, its aim is to keep the type system simple enough for domain experts—who are usually not Computer Scientists—to understand.

Semantic types thus are similar to classes in ontologies in that they capture domain concepts but not implementation issues. However, there are important differences. Classes in ontologies usually have a structure (this of course depends on the paradigm used to express the ontology)

which defines the class intensionally. By contrast, semantic types do not have such substructure. Semantic types can be defined extensionally by providing a sufficiently large set of expressions of this type. Thus, typed Asset Expressions can differ along two dimensions:

1. In *structure* by their abstractions and
2. in *type* by being of different semantic type.

This is depicted in figure 3.8 which shows the two independent dimensions. On the left, there is a small hierarchy of semantic types. For simplicity, the level in this hierarchy is indicative of the specificity of the semantic type. Likewise, the amount of abstractions indicates the number of abstractions in an expression, regardless of type and name. Thus, each dot represents in fact a set of Asset Expressions.

It is possible to predefine explanatory structures coupled with a semantic type. Such structures called *traits* (see section 4.1 for more details) can be helpful for recurring situations in which some user guidance is desired. However, as traits do not constitute an intensional definition of semantic types, they should be considered a suggested structure to build an expression for use in particular situations. Intensional definition of semantic types is avoided to obtain simplicity in the type system and flexibility in modeled entities.

3.3.2 Type Construction

Semantic types can either be simple types, which are defined by the creator of the expressions and are embedded in the type hierarchy, or function types, which are constructed from two semantic types by means of the $\rightarrow$ type constructor. Each type T has a corresponding list type constructed from this type by $*$ and written $\mathsf{T}*$. The syntax for types is:

$\mathsf{T} ::= \textit{primitive} \mid \mathsf{T} \rightarrow \mathsf{T} \mid \mathsf{T}*$

Function types can optionally be written in parentheses to enhance readability, i.e., $\mathsf{T}\rightarrow\mathsf{T} = (\mathsf{T}\rightarrow\mathsf{T})$. The typing rules of $\mathrm{AE}_{\rightarrow}$ follow examples from functional programming languages. Abstractions are typed in function types to indicate that an explanation is still missing to form a full description of an entity.

The type T of an Asset Expression e_1 is separated from the expression by a colon:

$\mathsf{e}_1 \colon \mathsf{T}$

In the following, capital letters will be used for arbitrary type names. Other types in examples will be written with capitalized first letter. All types save the built-in type Any have a single super-type. This relationship is written as <:, e.g.:

$\mathsf{T} <: \mathsf{S}$

Where T is more specific than S (i.e., S is the super-type of T). An expression of a subtype is substitutable for an expression of a supertype.

3.3.3 Expression Typing

Multimedial content and variables of abstractions are typed explicitly, for example the variable depicted and the image in the following expression:

$\mathsf{e}_1 := \lambda\mathsf{depicted}\colon \mathsf{Ruler}.$  $\colon \mathsf{EquestrianStatue}$

Thus only the syntax of abstraction and content have to be amended with type declarations:

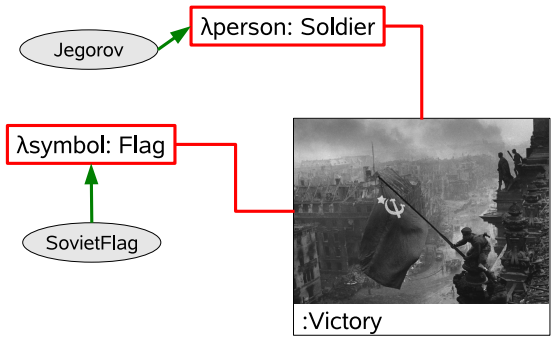


Figure 3.9: *Visual notation for typed Asset Expressions. Extensions are explicit typing of abstraction variables and content.*

λv: Type.e	for a variable v and an expression e as well as
C: Type	for a content C

All other syntactical elements remain the same. The type declaration ":" binds stronger than all previously introduced language elements. The visual notation is extended in a similar manner by introducing type annotations on abstraction variables and on content. The extensions are illustrated in figure 3.9 and table 3.6.

The types of expressions can be inferred by rules, which will be discussed in section 3.3.4. The expression e_1 which has an open abstraction is of function type:

e_1: Ruler→ EquestrianStatue

An expression e_2, which provides an application for the abstraction, is again of simple type:

$e_2 := (e_1)$Napoleon $\quad e_2$: EquestrianStatue

Given a previously defined expression Napoleon which is of type Ruler and thus matches the domain type of the function type. This allows for multiple explanations through pairs of abstraction and application without modifying the semantic type of the core. Different Asset Expressions that explain the same content in different ways will thus have the same semantic type, making them substitutable for each other if all abstractions are filled. Table 3.6 gives further examples of expressions and their type.

Existing expressions can be *lifted* explicitly to a more specific type, similarly to type casts in programming languages. The lifting of expression: S to a type T<:S is written as:

expression ↑ T

By means of a lifting the promise can be made to the type checker that the expression is actually of a more specific type than was originally modeled. If the type T that is lifted to is less specific that the original type of the expression, the lifting is ignored. This is feasible because of substitutability as will be shown in the next section.

All in all, the main goals of the type system are:

- *Making it harder to build non-sensical expressions.* The contribution of the type system to this end is to only allow applications if there is a matching abstraction and if the type of the abstraction variable matches the type of the expression that is to be applied. Analogous to the above example of expression e_2, the expression e_2' applying a common person would not be well-typed:

Expression	Type
 : Victory	Victory
λflag: Symbol  : Victory	Symbol→ Victory
λflag: Symbol  : Victory λauthor : Photographer	Photographer→ Symbol→ Victory
 : Flag λflag: Symbol  : Victory	Victory (Assuming Flag<:Symbol)

Table 3.6: *Examples of expressions and their type*

$$\mathsf{JoeDoe} := \ldots : \mathsf{Person}$$

$$\mathsf{e}'_2 := (\mathsf{e}_1)\mathsf{JoeDoe} \qquad \text{; error!}$$

Assuming Ruler is not a super-type of Person.

- *Capturing the nature of large expressions as a whole.* As a consequence of the typing rules for abstraction and application, the semantic type of an expression does not change regardless of the number of explanations employed. The overall semantic type also does not depend on the types of other expressions applied in explanations.
- *Relating expressions which describe similar real-world entities.* Another benefit of the semantic type's independence of structure is that expressions which describe similar entities in the real world are of equal type (or related by sub- or super-type relations).

3.3.4 Typing Rules

$\mathcal{S}$ is a set of simple semantic types which are defined by the user. These types, which are introduced by the user, are used to type content and variables. The semantic types in $\mathcal{S}$ can come from different domains.

A context Γ is constructed, which contains the semantic types of all variables and contents. The "comma" operator extends the context with a new type assertion. The grammar of the context is (ϵ is the empty context):

$$\Gamma ::= \epsilon \mid \Gamma, x : \mathsf{T} \mid \Gamma, C : \mathsf{T} \qquad \text{for variables } x \text{ and contents } C$$

The statement $\Gamma \vdash e\colon \mathsf{T}$ then means, that the expression e has the type T under the assumptions Γ. In typing judgements the expression e might be a complex expression, which is normally not assigned a type in Asset Expressions. Nevertheless, the "colon" operator is used to note its type. Typing judgements are presented in the form:

$$\frac{\text{premise}_1 \qquad \text{premise}_2 \qquad \ldots \qquad \text{premise}_n}{\text{conclusion}}$$

The context is used to look up the types of abstraction variables:

$$\frac{x\colon \mathsf{S} \in \Gamma}{\Gamma \vdash x : \mathsf{S}}$$

Asset Expressions are constructed from several pieces of (representations of) content C each of which is a semantic type $\mathsf{S} \in \mathcal{S}$.

$$\frac{C : \mathsf{S} \in \Gamma}{\Gamma \vdash C\colon \mathsf{S}} \qquad \mathsf{S} \in \mathcal{S}$$

Users can explicitly define subtype relationships between the semantic types. Expressions of type T are type-wise substitutable for expressions of transitive super-types of T:

$$\frac{\Gamma \vdash t : \mathsf{S} \qquad \Gamma \vdash \mathsf{S} <\colon \mathsf{T}}{\Gamma \vdash t : \mathsf{T}} \qquad \mathsf{S}, \mathsf{T} \in \mathcal{S}$$

Abstractions and applications are typed as usual in the λ-calculus. The type assertion in the conclusion ": $(\mathsf{S}\rightarrow\mathsf{T})$" refers to the whole expression $\lambda x.e$ and not just the body e. The expression is written in parenthese to emphasize this, it is not an application.

$$\text{Abstraction} \qquad \frac{\Gamma, x : \mathsf{S} \vdash\ e : \mathsf{T}}{\Gamma \vdash (\lambda x.e) : (\mathsf{S}\rightarrow\mathsf{T})}$$

$$\text{Application} \qquad \frac{\Gamma \vdash e : \mathsf{T}\rightarrow\mathsf{S} \qquad \Gamma \vdash f : \mathsf{T}}{\Gamma \vdash (e)f : \mathsf{S}}$$

As name definitions evaluate to the expression the name is bound to, they are also of this type. Again, the expressions in the conclusion are written in parenthese to emphasize the scope of the typing annotation for the purpose of type inference rules.

$$\frac{\Gamma \vdash e : \mathsf{S}}{\Gamma \vdash (x := e) : \mathsf{S}}$$

Lists of Asset Expressions are of the list type of the common super type of all contained expressions:

$$\frac{\Gamma \vdash e_i : \mathsf{S}}{\Gamma \vdash \{e_1, \ldots, e_n\} : \mathsf{S}*} \qquad i \in [1, n]$$

Therefore, there can be several valid types for lists. If a hierarchy of semantic types defines $S <\colon T <\colon U$ and an expression of type $T*$ is required, expressions of types $S*$ or $T*$ will suffice. This is possible because lists are immutable.

A lifting $\uparrow$ can explicitly change the type of an expression to a more specific one. However, a lifting to a type that is more general than the type of the expression has no effect.

$$\frac{\Gamma \vdash e\colon \mathsf{S} \qquad \Gamma \vdash \mathsf{T} <\colon \mathsf{S}}{\Gamma \vdash (e \uparrow \mathsf{T}) : \mathsf{T}} \qquad \frac{\Gamma \vdash e\colon \mathsf{S} \qquad \Gamma \vdash \mathsf{S} <\colon \mathsf{T}}{\Gamma \vdash (e \uparrow \mathsf{T}) : \mathsf{S}}$$

As noted previously, the rebinding of names can implicitly change expressions which make references to this name. If the newly bound expression is an extension or correction of the previously bound, such a change will not modify the semantics of the referencing expression, but will merely make the explanations more detailed. If, however, the newly bound expression is entirely different, the referencing expression can become wrong or meaningless in many ways. Typed Asset Expressions can remedy this situation by distinguishing two cases of rebinding based on the type of the new expression:

$$\text{Variables}\quad \frac{x\colon \mathsf{S} \in \Gamma}{\Gamma \vdash x : \mathsf{S}} \tag{3.1a}$$

$$\text{Content}\quad \frac{C : \mathsf{S} \in \Gamma}{C\colon \mathsf{S}} \qquad \mathsf{S} \in \mathcal{S} \tag{3.1b}$$

$$\text{Substitutability}\quad \frac{\Gamma \vdash t : \mathsf{S} \quad \mathsf{S} <\colon \mathsf{T}}{\Gamma \vdash t : \mathsf{T}} \qquad \mathsf{S}, \mathsf{T} \in \mathcal{S} \tag{3.1c}$$

$$\text{Abstraction}\quad \frac{\Gamma, x : \mathsf{S} \vdash e : \mathsf{T}}{\Gamma \vdash (\lambda x.e) : (\mathsf{S} \to \mathsf{T})} \tag{3.1d}$$

$$\text{Application}\quad \frac{\Gamma \vdash e : \mathsf{T} \to \mathsf{S} \quad \Gamma \vdash f : \mathsf{T}}{\Gamma \vdash (e) f : \mathsf{S}} \tag{3.1e}$$

$$\text{Naming expressions}\quad \frac{\Gamma \vdash e : \mathsf{S}}{\Gamma \vdash (x := e) : \mathsf{S}} \tag{3.1f}$$

$$\text{Lists}\quad \frac{\Gamma \vdash e_i : \mathsf{S}}{\Gamma \vdash \{e_1, \ldots, e_n\} : \mathsf{S}*} \qquad i \in [1, n], \mathsf{S} \in \mathcal{S} \tag{3.1g}$$

$$\text{Lifting}\quad \frac{\Gamma \vdash e\colon \mathsf{S} \quad \mathsf{T} <\colon \mathsf{S}}{\Gamma \vdash (e \uparrow \mathsf{T}) : \mathsf{T}} \quad \frac{\Gamma \vdash e\colon \mathsf{S} \quad \mathsf{S} <\colon \mathsf{T}}{\Gamma \vdash (e \uparrow \mathsf{T}) : \mathsf{S}} \qquad \mathsf{S}, \mathsf{T} \in \mathcal{S} \tag{3.1h}$$

Figure 3.10: *Semantic typing of Asset Expressions*

1. If the newly bound expression is of the same or a substitutable type, binding remains a one-step operation and is always allowed.

2. If the newly bound expression is of a different, non-substitutable type, binding becomes a two-step operation composed of an unbinding of the name and a subsequent binding with the new expression. Note that the unbinding is only possible if the name is not referenced anywhere.

The effect of these rules is that rebindings of names which have dependent expressions can only be carried out if the new expression is of compatible type. This can still modify the semantics of the dependent expressions, but the equality of expressions cannot be determined beyond their type (also see section 4.4).

3.3.5 Rationale of This Type System

The content in Asset Expressions is interpreted by the user in the context of the supplied explanations and the user's contextual knowledge. This interpretation results in a real-world entity, *not* in its medial representation. The return type of functions, which provide such explanations, is therefore a semantic type. The formal parameters are whatever the creator of the expression chose them to be.

At a syntactic level $\text{AE}_{\to}$ needs to achieve closedness under composition. If one thinks of content as a function, the results of function applications can again be used as parameters in new applications. Specifically, it is necessary that one is able to use existing expressions to supply explanations in new expressions. This can be achieved by proper typing of abstraction variables to allow expressions of the right kind to be applied.

At a semantic level, one observes that the way a real-world entity is explained does not change its nature, thus the type should be the same. This has two consequences: (1) the abstractions and applications used to explain the entity must not manifest themselves in the type of the overall expression, and (2) several explanations of a consistent view on the same entity have the same type. The first seems suprising initially as many modeling approaches

handle this differently. Take object-oriented modeling as an example, where the details of the explanation (i.e., the object's attributes) are part of the class. However, such elision makes sense if one considers one's own mental model of such a type: Once one has understood a certain entity, the particular explanation that led to this understanding might well vanish.

Just as with functions, one cannot draw conclusions from the formal parameters of the function to the nature of the function as a whole or its (semantic) return type. A function with parameters int × int is not necessarily an addition and might have any return type. Similarly, the types used in explanations cannot be mapped onto a semantic type for the whole expression. This is revised in section 7.1.

The second requirement is reasonable as composition of expressions is highly desired. It is fulfilled by the typing rules for application and abstraction which allow the semantic type of an expression not to change with additional explanations.

Each variable of an abstraction is assigned a semantic type. It restricts the expressions, which can be applied to fill this abstraction, as the expressions must be of a substitutable semantic type. This use of semantic types can also be found in hypermedia systems [BVA+97], where such types are used to restrict the kind of document a link can point to.

Just as with ordinary functions, the semantic type of Asset Expressions depends on the type of the function's body (often content) if all abstractions have full applications. If this is not the case, the total expression is of a function type. This is desirable when composing expressions as an only partially explained expression is hardly fit for use in further explanations. The function type then prevents its use in places where a simple type (i.e., an expression with complete explanations) is required for the explanation.

3.3.6 Typing Example

Consider the following typed Asset Expressions as an example:

Napoleon :=  : Emperor

$e_{1,typed}$:= (λequestrian: Ruler.  : EquestrianStatue)Napoleon

Two semantic types are used: Ruler and EquestrianStatue. They are part of the type hierarchy and might have been specified by the creator of this expression. Alternatively, they could be imported. However, as they are in the default namespace, they are part of the current application domain.

Using the type inference rules shown in figure 3.10, the well-typedness of $e_{1,typed}$ can be established.

$$\dfrac{\dfrac{\dfrac{}{\vdash \text{  } : \text{EquestrianStatue}}}{\vdash \lambda\text{equestrian: Ruler.}\text{  } : \text{Ruler}\rightarrow\text{EquestrianStatue}} \qquad \dfrac{\dfrac{\vdash \text{Napoleon: Emperor}}{\vdash \text{Napoleon: Ruler}}}{}}{\vdash (\lambda\text{equestrian : Ruler.}\text{  } : \text{EquestrianStatue})\text{Napoleon: Emperor}}$$

Chapter 4

Extended Asset Expression Language

Asset Expressions provide a high degree of freedom to model real-world entities, examples of this can be found towards the end of this chapter and in chapter 8. Creators of expressions can describe these entities in any way that suits them. In particular, the semantic type assigned and the abstractions used are totally independent.

4.1 Traits for Asset Expressions

Total independence of explanations and semantic types is certainly desirable in general to allow for expressions of personal opinion (also see section 4.6.2 on openness and dynamics). Under some circumstances—for example when an application domain has been understood to a high degree—such complete freedom may not be necessary any longer.

To introduce a dependency between the level of abstraction and specificity of semantic types, i.e., to only allow a certain area in figure 3.8 to be reached, *traits* can be used. Traits for Asset Expressions serve as expression templates which prescribe a number of abstractions (possibly none) and a semantic type.[1] A trait for Asset Expressions is defined as follows:

trait name [**refines** super-trait] **of** semantic-type **with** abstractions

Traits have a name and define a semantic type and a set of abstractions. Traits can have a super-trait whose abstractions are inherited to the sub-trait. The additional abstractions specified with the sub-trait are added to those specified with the super-trait. Expressions created through a trait have the semantic type specified in the trait's definition. If the super-trait is EmptyTrait, the super-trait declaration may be omitted. EmptyTrait is a built-in trait that defines no abstraction and the semantic type Any. Expressions are created based on traits as follows:

create trait *expression*

Which creates a new expression based on *expression* by adding the abstractions provided in the trait. *expression* is also lifted (see section 3.3.4 on lifting) such that the complete expression is of the semantic type prescribed in the trait.

In the domain of art history it might be sensible to define a trait painting:

[1] The term "trait" is borrowed from object-orientation (see, e.g., [AC96, pp 47, 73]) where traits contain methods and serve as prototypes for concrete objects.

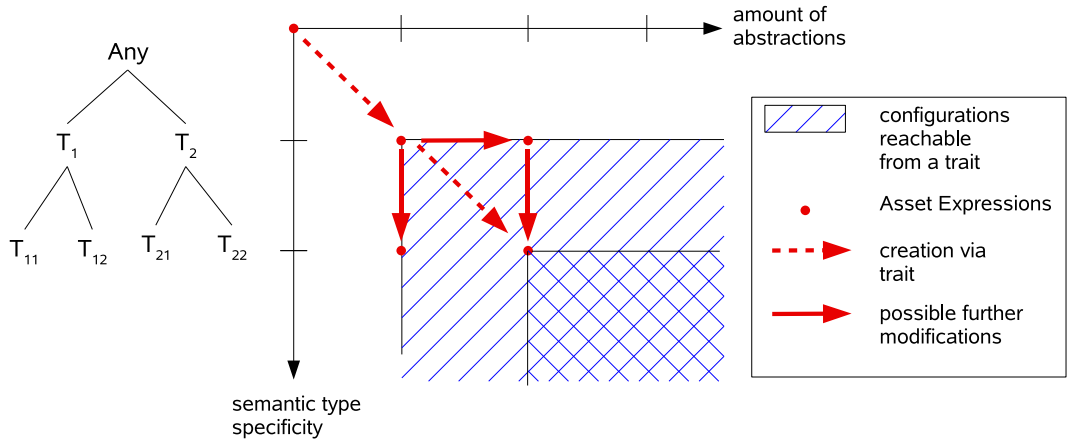


Figure 4.1: *Traits prescribe a number of abstractions and a certain semantic type. By creating a new expression from a trait (instead of from scratch) not all areas in the abstraction-semantic type plane are reachable anymore, compare to figure 3.8 where no such restrictions are in place.*

trait painting **refines** EmptyTrait **of** WorkOfArt **with** λpainter: Artist.λpaintedIn: Epoch

Thus requiring that any expression created through this trait will be a WorkOfArt and have at least abstractions typed as Artist and Epoch. A particular painting can then be created by:

create painting  =λpainter: Artist.λpaintedIn: Epoch.  : WorkOfArt

The expression created by means of the trait has open abstractions (as defined in the trait), which can now be met by corresponding applications. These applications are not part of the trait because they would only be useful to create a single expression. An expression created via a trait is equivalent to its manually created counterpart, thus applications are made with the same syntax:

David := ...

e_{SB} := (**create** painting )David

Resulting in an expression e_{SB}, which has one open abstraction:

e_{SB} = (λpainter: Artist.λpaintedIn: Epoch. )David

Several traits can be used together for the creation of an expression. The resulting expression has all abstractions prescribed by the traits and is of the most specific semantic type.

trait portrait **refines** EmptyTrait **of** WorkOfArt **with** λdepicted: Person

$e_{extended}$:= (((**create** portrait **create** painting )Napoleon)David)Renaissance

Traits restrict the possible configurations (i.e., combinations of abstractions and semantic type) of expressions by introducing obligations for explanations. While an expression created

through a trait can be further abstracted from, any abstractions introduced by the trait cannot be removed. Similarly, the semantic type can be lifted to a more specific type, but a semantic type more general than prescribed by the trait is no longer reachable. As shown in figure 4.1, a trait (dashed arrow) therefore limits possible configurations to the striped area. More detailed configurations can be reached by modification (as indicated by the solid arrows in the figure) inside the striped area. The figure also shows a second use of a trait. This second use further limits the possible configurations to the intersection of the areas.

4.2 Asset Expression Query Language

Asset Expressions can be used to create networks of multimedial content, which serve to describe entities. A large database of such expressions can serve to describe an application domain. While some insights might already be gained during the modeling process, users will generally want to later retrieve their expressions. Named expressions can be retrieved by their name. However, this is only possible if the name is remembered. Of course, the desired expression has to have a name in the first place. This will not generally be the case as the purpose or objective of work can have changed since the expression was created. Thus, at creation time the expression might not have been deemed interesting enough to be given a name. Instead, the creator might have chosen to give a name only to a larger expression that includes the desired one.

To allow for more flexible retrieval of expressions, the Asset Expression Query Language (AEQL) is introduced. The key features of this language are:

1. It enables users to select any part of an expression. The set of parts of an expression is the set of all its transitive subexpressions. Thus, not only named expressions can be retrieved but also any other Asset Expression.

2. The results of all queries are again well-formed Asset Expressions. This is important to provide flexible recombinability of results to create new expressions.

3. Particular expressions can be selected from a larger body of expressions based on their values, on their type, as well as on their structure.

4. New Asset Expressions can be built in the language. This is related to feature 2 in that it can be desirable to construct a new expression around the results of a query.

In general, the second feature is a lesson leant from existing query languages in various fields of application. In the Structured Query Language SQL [Sta03], for example, the results of queries are generally relations (just as the inputs), resulting in, e.g., facilitated combinability of both query results and query language constructs. Conversely, XPath as another example allows many queries whose results are not proper XML documents. The language also does not provide the means to transform the results into such documents. This can make the use of the language considerably more difficult in situations such as view definition [Bos04].

There are several approaches to queries on λ-terms in the literature. When searching for expressions given some selection criteria, one can generate an exhaustive list of expressions which match these criteria [Kat05]. Selecting matching expressions from a set then is a trivial task, one only has to compare for equality. The search criteria and the structure of the expressions to compare against have to be constructed in a way that ensures that the exhaustive list of potential candidates is finite. When the domain of the search are, e.g., small functional programs [Kat05], this is the case. To query different domains, these domains can be expressed in λ-expressions [HKM93] and a query language can be constructed on these expressions or a usual query language for the domain can be embedded into the λ-calculus.

The query language presented here takes a more conventional approach. It is a special purpose language in the sense that its language elements are not λ-terms. Given the complexity

of the expressions resulting from the embedding in the λ-calculus [HKM93], such an embedding is not feasible for queries on Asset Expressions where the intent is to provide domain experts with a suitable query language. Indeed, for some users the AEQL presented below might still prove too complex.

AEQL is not designed with particular query scenarios in mind. Rather, it is intended as a general query language for Asset Expressions, onto which more specialized scenarios with special languages can be mapped. Its design is based on previous experiences with querying, in particular in the area of semi-structured data [BSG+04, Bos04]. Examples of applications include the matching of signatures by similarity (see section 7.1) and the creation of conceptual models based on a body of Asset Expressions (discussed in section 7.2).

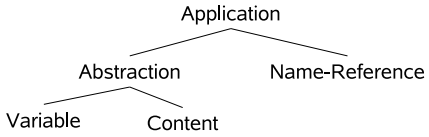
The Asset Expression Query Language aims to provide query facilities at Asset Expression level. In other words, it is neither concerned with querying multimedial content as multimedia query languages (e.g., [LÖSO97]) nor with information retrieval from medial content (e.g, [CMF96]). Multimedia query languages allow queries against technical properties of the content such as the playing time of a video or the dimensions/colors of an image. They do not address the aspects of interrelation of content instances or the further explanation of particular parts of a content. In a survey conducted on queries of users against an archive of images, it was found that the type of query depends on the users' background and their motivation to run the query [Kei94]: While more graphically inclined users (e.g., someone working on illustrational graphics) might ask for images of some object or images in which a certain color is dominant, other users base their queries on background and contextual information. While AEQL leaves graphical properties to multimedia query languages, it does address the other concerns. However, for a particular application domain a specially targeted language can be more appropriate, just as users of information systems which are based on a relational database are not expected to write SQL queries.

4.2.1 Navigating Expressions

Asset Expressions can be represented as a tree structure. This representation is central to AEQL. Each expression has zero (variables, content), one (name definitions), or two (application and abstraction) subexpressions. As an example, consider the expression:

$e_1 := (\lambda x.C)A$

Which can be quite conveniently represented as its abstract syntax tree:



Each sub-tree of this tree is again a well-formed Asset Expression by construction, see the full grammar presented in section 4.7. Thus, each sub-tree (i.e., each sub-expression) might be of interest to users wishing to construct new expressions or educate themselves about existing ones. Therefore, AEQL includes means to navigate the tree to a particular expression of interest. The edges of the tree are labeled according to the nature of the relationship of the parent and the child as shown in figure 4.2.

A labeled edge is called an *axis*[2]. Along an axis one can navigate from parent to child expression. An axis is applied to its *context* by a "/". The context can be a single expression or a list of expressions. Using the expression e_1 from above:

$e_1/\text{operator} = \lambda x.C$ $\quad$ $e_1/\text{operand}=A$

[2]This is not the same as an axis in XPath even though the syntax is similar.

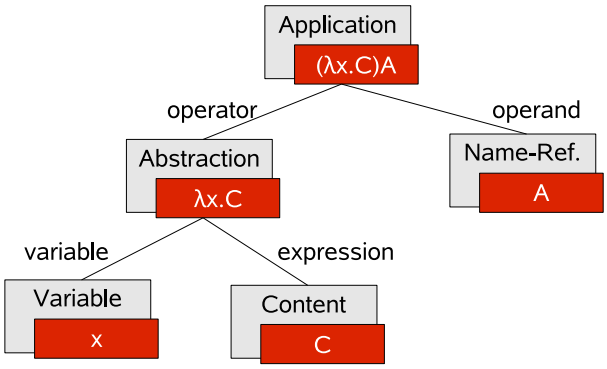


Figure 4.2: *Abstract syntax tree representation of the Asset Expression (λx.C)A together with concrete syntax for every node. The edges are labeled with the relationship of parent and child to allow navigation in the query language.*

For convenience, there is a special context "_" which is a list of all currently defined expressions.

Navigating the axis yields a list of expressions, which are obtained by navigating the axis individually for each expression in the context and putting the resulting expression in the result list. If the axis is not applicable to an expression in the context, nothing is put in the result for this expression. However, this is not an error.

Axes can be applied repeatedly and in combination by concatenating them, each axis application is called a *step*. The first axis is interpreted in the context given. Its result is used as context for the second axis and so on.

e_1/operator/expression = C

As indicated above, the initial context as well as intermediate contexts (i.e., results of previous axes) can be lists of Asset Expressions:

e_2 := (λy.(λz.C_2)A)B

{e_1, e_2}/operator/variable={x, y}

The available axes are summarized in table 4.1. The first four axes in table 4.1 filter based

Axis	Applicable to	Matched expression
/variable	abstraction, $\lambda v.e$	variable v of an abstraction
/expression	abstraction, $\lambda v.e$	expression e over which was abstracted
/operator	application, $(A)B$	expression A that is applied
/operand	application, $(A)B$	expression B which is applied to
/abstraction	all	subexpressions that are abstractions
/application	all	subexpressions that are applications
/content	all	subexpressions that are content
/parent	all	parent expression
/*	all	any kind of expression, useful in combination with predicates (see section 4.2.2)

Table 4.1: *Axes available in AEQL and the corresponding expressions.*

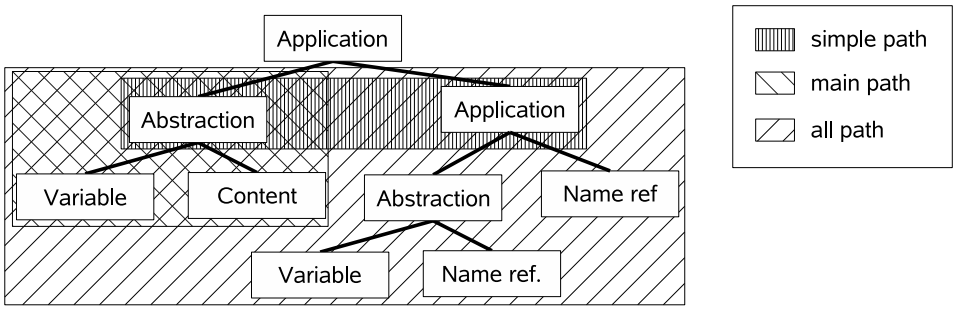


Figure 4.3: *The subexpressions included by different paths relative to the topmost application. The simple path "/" includes just the direct subexpressions. The main path "//" includes all transitive subexpression except those only reachable via the operand of an application for axis evaluation. The all path "///" includes all subexpressions.*

Path	Name	Description
/	Simple	Only direct subexpressions are included for evaluation.
//	Main	Transitive subexpressions are included except those which can only be reached via operands.
///	All	All transitive subexpressions are included.

Table 4.2: *Paths of AEQL*

on the relationship of parent and sub-expression. Another set of axes is available which filters based on the kind of expression in the context. These axes are /abstraction, /application and /content which filter for abstractions, applications, and contents, respectively. The /parent returns the parent expression for each expression in the context. Additionally, there also is an identity axis which will pass on the entire context. Its purpose will become apparent once further elements of AEQL have been introduced.

Up to now, an axis is only evaluated against the direct subexpressions. This can be inconvenient, for example if access to all abstraction variables in an expression is desired. One would then have to supply the appropriate combination of /operator and /variable axes to pick all those variables out of the application. The resulting query would heavily rely on the structure of the expression which was envisioned when the query was created. To facilitate queries that deal with deeply nested expressions, *paths* determine against what subexpressions an axis is evaluated. The already introduced *simple path* "/" evaluates the corresponding axis against the direct subexpressions. The *main path* "//" evaluates against all transitive subexpressions except for those which are only reachable via an operand of an application. Furthermore, there is an *all path* "///" which includes all subexpressions. The paths are illustrated in figure 4.3. Table 4.2 lists all path types. Take the expression e_3 as an example :

$e_3 := (\lambda v.C)(\lambda w.C_1)A$

Evaluating e_3//variable results in v but not in w.

4.2.2 Predicates

From a given context, the path of a step puts a certain set of subexpressions into the context. This context is then filtered depending on the axis. To allow for expression selection by criteria other than axes, a predicate can be given after the axis in square brackets:

/a[*predicate*]

The predicate is evaluated against the context, which has already been filtered through the axis. By means of a predicate, abstractions can for example be further filtered for variables:

/abstraction[/variable=x: T]

Where it is required that the variable of the abstraction is x of type T. Equality of variables is defined as equality of variable name and type. The type can be omitted and is then ignored in the comparison. Individual predicate expressions can be negated or combined through conjunction as well as disjunction. The following AEQL expression matches all expressions that (1) are abstractions over variable x and (2) contain the content C anywhere in the body of the expression:

/abstraction[/variable=x and ///content=C]

Equality of two contents is defined as the equality of what they "show", e.g., the pixels of an image or the words of a text. As in the case of variables, the semantic type of the content can also be given. Implementations need to handle this correctly for each technical content type. Besides comparisons, predicates can also be defined on the type of an expression. The following predicate checks for the type of the variable of an abstraction:

///abstraction[/variable: EquestrianStatue]

Some functions are provided such as match-abs and match-app. The former retrieves the abstraction matching the application given as argument, the latter works in the opposite direction. They can also be used in predicates. An example of their use will be presented in section 4.5. For further examples, the expression e_4 is defined:

e_4 := ((λx.λy.C)(λa.D)E)(λb.F)G

To retrieve all applications containing F:

e_4///application[///content=F]={(λb.F)G}

Find all expressions which contain a variable a:

e_4///*[///variable=a]=
{ ((λx.λy.C)(λa.D)E)(λb.F)G, (λx.λy.C)(λa.D)E, (λa.D)E, λa.D }

The result contains many expressions which are subexpression of another expression in the list. When this is not desired, the result can be filtered for such duplications by means of the function nosubs:

(nosubs)e_4///*[///variable=a]={ ((λx.λy.C)(λa.D)E)(λb.F)G}

The full syntax of AEQL can be found in section 4.7.

4.2.3 Creation

New expressions can be created in conjunction with queries using the ordinary syntax for Asset Expressions. To introduce an additional explanation on an expression retrieved via a query, one could for example write:

F := ... G := ...
e_5 := (λartist: Painter._//application[///*=F])G

This expression retrieves all applications (context "_") which contain the expression F. Over these applications an additional abstraction is created and a matching expression is applied.

It is also possible to use the results of several queries to create new expressions. This can be interesting to combine existing expressions in new ways, for example to summarize them. First of all, a new expression can be created for each expression in a list of expressions (resulting in a new list):

foreach b **in** _///*[.: Book] **return** λauthor: Person.b

Resulting in a list of expressions based on the query, but *each* with an additional abstraction. **foreach** variable **in** *list* takes a list of expressions and consecutively binds each to the variable. With this binding the expression given in **return** *expression* is evaluated for each binding. The result is therefore a list with the same number of expressions as in the original list. This list does not have to be obtained through a query, it could also be given directly or as a reference to a named expression. Filtering the input list can be achieved by an additional query with a predicate on this list.

It is possible to introduce additional variable bindings with further uses of **foreach**. All variables are then available in the return expression. Thus the return expression now needs to be evaluated for each element in the cartesian product of the input lists. To attach all persons as (single) authors to all books, one could write:

foreach b **in** _///*[.: Book]
  foreach a **in** _///*[.: Person]
  return (λauthor: Person.b)a

Making everybody author of every book is usually not desired. Just like query contexts can be filtered after each step, the cartesian product can also be filtered by a predicate:

foreach b **in** _///*[.: Book]
  foreach a **in** _///*[.: Person]
  where b **in** a//abstraction [/variable= book]
  return (λauthor: Person.b)a

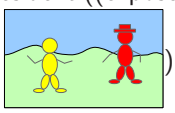
4.3 Handling Components

Components define parts of a content through selectors. The components of a content are internal to this content and not directly accessible to Asset Expressions based on the content. To make the components accessible, dedicated access means are available which define the connection of the component and the expression based on it. One such means has already been introduced above (page 52): Using expose components of a content were made available to abstractions. In this section additional access means are introduced. These means are called component handling, of which there are three types: the mention handling through expose, the use handling through hole, and the piece handling through cut.

4.3.1 Mention-Handling

expose makes a named component available in such a way that the part of the content which is addressed by the component's selector is subject to the expression built on the component. When a component is made available to an abstraction using expose, the explanation given by the abstraction and its associated application talk about the content of the component. Using a modified version of the above example:

(λpresident.((expose)"JD": Name)(((child)"JD": Name)<coords, ...>)



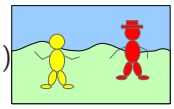
)PresidentOfAtlantis

Creating an expression which talks *about* the selected component (the person without hat). The default handling of a component is expose. This means that all abstractions which are made over the whole of the content (because no component is given) use expose handling.

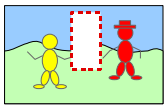
4.3.2 Use-Handling

Components can also be used to refer to a place in the content. In this scenario, the actual content at this place is unimportant, the place itself is of interest. Making available a component through expose *mentions* the piece of content to the associated explanation. Using hole *uses* the piece without regard to what actually is at that location (compare the general discussion on "use vs. mention" [Gar65]).

((hole)"JD": Name)(((child)"JD": Name)<coords, ...>)



=

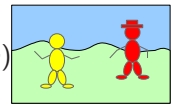


Components handled by hole are shown with white background in the visual notation.

4.3.3 Piece-Handling

The previous two handlings keep the component in the context of its content. Where this is not desired, the component can also be removed from its surroundings. This is achieved by the cut handling.

((cut)"JD": Name)(((child)"JD": Name)<coords, ...>)



=

This handling is one of the reasons why selectors are required to select a part of the content that can be a new content in its own right.

4.4 Normal Form with Content

Expressions in the λ-calculus can be programatically reduced to a normal form. This normal form plays an important role in determining the semantics of expressions: Two expressions which reduce to the same normal form are equivalent. The possibilities and issues of applying a similar approach to Asset Expressions are explored in this section.

4.4.1 Reduction

Reduction in the λ-calculus is essentially an operation which relies on the proper substitution of the operand of an application into the operator. The reduction ⇒ of an expression (λx.P)Q is defined as:

$$(\lambda x.P)Q \Rightarrow [Q/x]P$$

Where [Q/x]P is the substitution of Q for all occurrences of x in P. Rules to carry out these substitutions are defined in section 2.2.1.

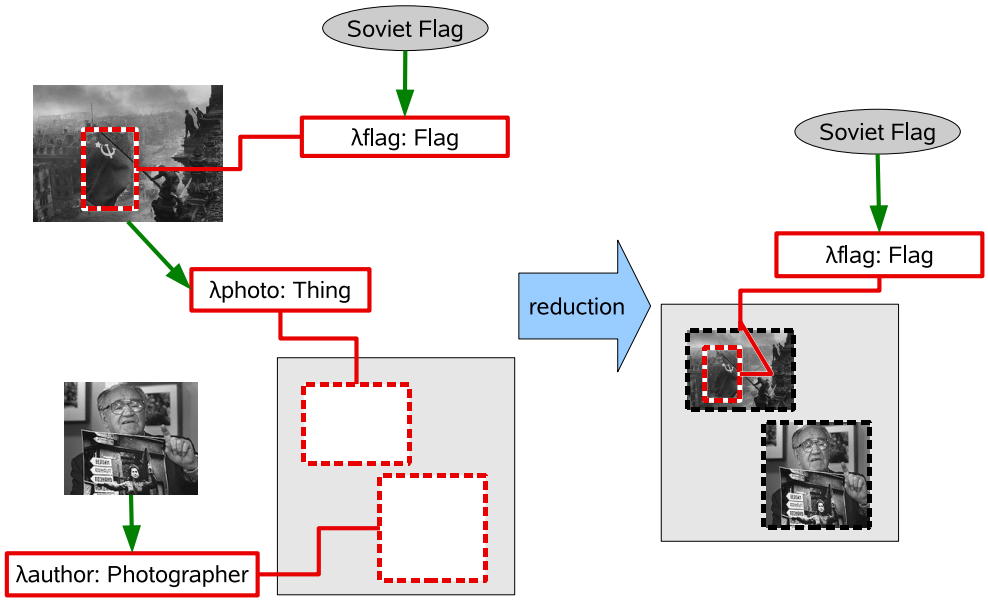


Figure 4.4: *Example of use-handling for content construction by reduction. The piece of the flag in the image is selected using two chained selectors.*

Asset Expressions can only be handled by the substitution rules of the λ-calculus in the special case that the operator in the reduction does not contain content. Not all expressions can be covered by substitution rules, essentially because—in the case of Asset Expressions—operand and operator generally can only be evaluated by a human. However, reduction requires this interpretation to be machine implementable. Whether an application in Asset Expressions is machine reducible, depends on the component handling used to bind the abstraction to the component: cut is always reducible, hole is in some cases. The cases that allow reduction in Asset Expressions are covered in more detail in the next section.

4.4.2 Substitution

Whether a substitution $[Q/x]P$, is possible depends on the expressions P and Q but can also depend on the component handling which is used for the component x. Programmatic substitution is possible in some cases:

1. Expressions that are plain λ-calculus expressions. These are expressions that do not use content at all or that do not perform any abstractions over content. Examples are:

   $\lambda a{:}\ T_1.(\lambda x{:}\ T_2.x)a, \qquad (\lambda a{:}\ T_1.a)$ 

   Substitution can be performed with the normal means of the λ-calculus. Content without abstractions (as shown in the second expression) is treated like a literal value in the λ-calculus. However, not using any content is not a very common case for Asset Expressions.

2. Use-handling is employed for the affected component. Two different cases of this are illustrated in figure 4.4. By use-handling the abstraction refers to the place in the content which is addressed by the selector of the component.

		operand			
		image	text	audio	video
operator	image	x	x		
	text		x		
	audio		(x)	x	
	video	x	x	(x)	x

Table 4.3: *Compatibility of substitution of content kinds into each other, which determines whether an application to a component handled by hole will be reducible. Text into audio requires non-trivial conversions. Audio into video is only possible if the component is selected appropriately (i.e., a time interval).*

If a plain content Q is applied to this abstraction (as shown in the example of the λauthor: Photographer abstraction in the figure), i.e., the substituted expression is a plain content, the substitution can be carried out. The applied content can be substituted into the content that is abstracted over, provided it is of the same kind. In this case the substitution is simply

$$(\lambda \mathsf{x}.\mathsf{P})\mathsf{Q} \Rightarrow [\mathsf{Q}/\mathsf{x}]\mathsf{P}$$

regardless of the structure of P. The square brackets are not used as an abbreviation for expose in this section but for substitution as in section 2.2.1. Some scaling operations might be necessary. If the content is not of the same kind, conversions are possible for some types of content. Table 4.3 shows an overview. Substituting text into audio and audio into video requires major conversion efforts in the case of text and a selector with appropriate addressing (a sufficient span of time) in the case of audio.

If Q has further explanations (as is the case with λphoto: Thing in figure 4.4), these are handled by chaining selectors. Q and P then take the forms

$$\mathsf{Q}{=}\ldots(((\mathsf{child})n_1)s_1)\ldots(((\mathsf{child})n_N)s_N)\ldots C_Q\ldots$$

$$\mathsf{P}{=}((\mathsf{hole})n_P)(((\mathsf{child})n_P)s_P)C_P$$

With C_Q and C_P some Asset Expressions. Additional components are introduced in the content of P. Their selectors are obtained by chaining the selector of the original component x in Q with the selector of the component of Q:

$$[\mathsf{Q}/\mathsf{x}]\mathsf{P}{=}\ldots(((\mathsf{child})n_1)s_P\ s_1)\ldots(((\mathsf{child})n_N)s_P\ s_N)[\mathsf{Q}/\mathsf{x}]C_P\ldots$$

The component handling and the abstractions are carried over from P. If reduction is carried out in normal order[3], reductions might be possible for the new components with chained selectors. The restrictions of content kinds as in the simple case also apply here.

3. If piece-handling is used, the part of the content addressed by the selector of the component is removed from its surrounding content. The result of this operation is not necessarily just a plain piece of content because abstractions over this component or any of its children remain in place. Abstractions over other parts of the content outside the handled component are not part of the resulting expression.

[3]In normal order reduction the leftmost application, i.e., the first redex, is reduced first [Rev88].

4.5 Content Construction

It is common for information systems to not just store entities, but to support the creation of new entities by recombination of existing ones as well. Means to interrelate Asset Expressions by creating new ones from the parts of existing expressions are required to provide this functionality. In section 4.2 the query language AEQL was presented, which offers means to select parts of expressions based on various aspects of these expressions and constructors to create new expressions from these selected parts. As Asset Expressions aim to provide an integrated representation of multimedial content and its conceptual description, such construction must also be possible for content. In fact, as all means to do so are already in place, it is the purpose of this section to given an overview of how to employ them for content construction.

Consider the scenario of figure 4.4, where the goal is to produce a tableaux for each photograph in the system that shows the photograph together with its author. The figure shows one example of this. The first step is the construction of a template with components for the photographer and the photograph. These components are made available to abstractions by use-handling. The template is used for each photograph. The photograph is applied to the template and the photographer is retrieved by following the application on the photograph which matches the abstraction for the photographer.

```
Template := λphoto: Thing.((hole)"person")(((child)"person")<coords-rel, ...>)
    λauthor: Photographer.((hole)"photo")(((child)"photo")<coords-rel, ...>)C
foreach p in (nosubs)_///*: Photograph
    return ((Template)
        p)(match-app)p//abstraction[./variable: Photographer]/operand
```

C is a canvas, such as an empty bitmap, onto which the two contents are layed out. This application of Asset Expressions can be used to aggregate superimposed information [DM99]. The superimposed information approach works by collecting information from several documents (which could be modeled in several Asset Expressions) and aggregating this information in a new, usually more concise, view.

In the same domain a different example might be concerned with the enrichment of descriptions. Consider the following scenario: Photographs are explained with the name of their photographer as a simple string, but richer models of persons are available which include the name. Expressions for the photographs shall be provided to an audience for whom the name of the photographer does not suffice but a more elaborate model is required. The photographer is therefore added with an additional λauthor abstraction to the photograph:

```
foreach g in _//*[.: Photograph]
    foreach p in _//*[.: Person]
    where p//application[./operator/abstraction/variable=photographer]
        /operand=a/application[./abstraction/variable=name]
    return (λauthor: Photographer.g)a
```

4.6 Notes on the Use of Asset Expressions

When creating Asset Expressions to model real-world entities, the question when to model an entity as a named expression of its own and when to include it in the description of another entity often arises. This question, which is essentially about the point of focus of the application domain, also occurs in conceptual modeling in general [BMS84, BJJW97]. Because Asset Expressions are not constrained by a general schema, the question needs to be answered by the users based on their experience on a per-instance basis. Some arguments to consider when evaluating the separation of an entity include:

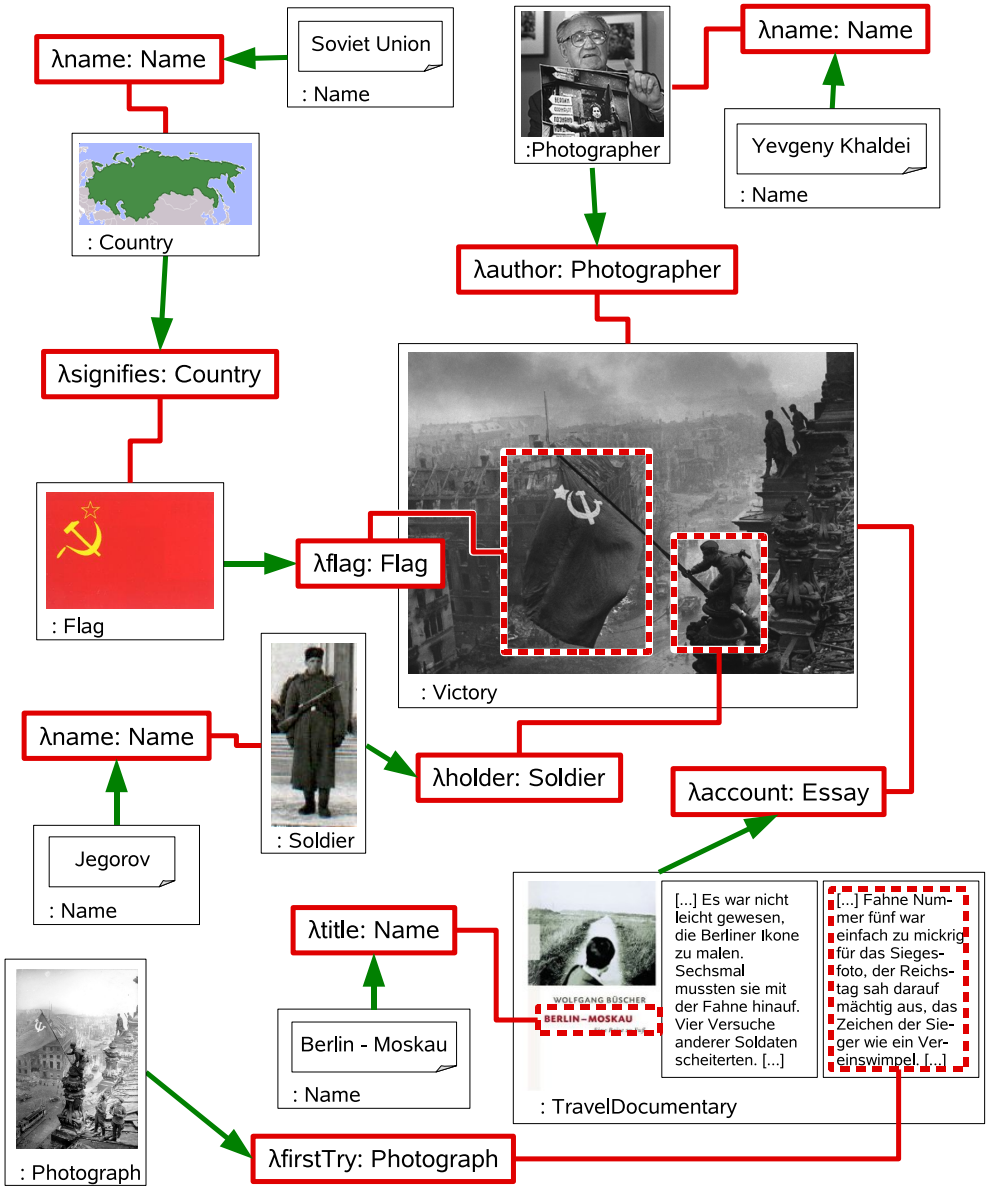


Figure 4.5: *Example of a large Asset Expression network which explains substantial parts of the context of its content.*

- *Domain semantics.* Entities that take an active role in the application domain or that serve to interconnect other entities are good candidates for modeling in dedicated expressions.
- *Reuse.* If the entity is needed more than once in the same form, the expression to model this can be reused. However, if the same entity reoccurs but is required in different forms (i.e., with different sets of explanations) this is not necessarily a case of reuse for Asset Expressions.

Reuse will often be caused by domain semantics, but even if domain semantics suggest reuse, the necessity of diverse explanations will sometimes undo the benefits. However, in cases of

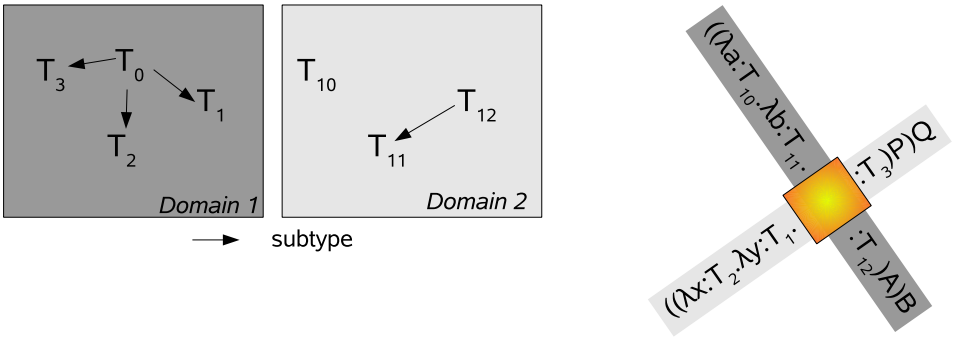


Figure 4.6: *Example of typing with semantic types from multiple domains.*

diverse explanations a common kernel can sometimes be found in all expressions modeling one entity. This kernel can then be pulled out and modeled in an expression of its own. It connects the related expressions.

A similar issue was already encountered in Hypertext systems when dividing a document into chunks [TW86]. A too small chunk size (corresponding to very many independent Asset Expressions) causes fragmentation of the domain and makes organization difficult. A too large chunk size often prevents users from expressing a single idea in this chunk.

4.6.1 Multiple Domains

Most pieces of content can be interpreted in different application domains. A common example of two such domains are (1) the domain of the creation of (or *work* on) the content and (2) the domain of its statement or meaning. The first domain might be concerned with painting techniques, canvases, or brushes in the case of a painting. The second might deal with religion or political iconography depending on the message of the actual painting. As the content is a representation of a single real-world entity, this representation is of relevance to both domains and should be explained appropriately in each. In the work domain the content might be typed as OilPainting, while in the political iconography it might be necessary to type it as an EquestrianStatue.

Unfortunately, each expression must have one single type which is from a single-inheritance hierarchy. It would clearly be incorrect to make OilPainting a subtype of EquestrianStatue or vice versa as this is wrong in both application domains. A solution to this dilemma is shown in figure 4.6. The same content is typed with different semantic types in two expressions for each domain. Such multi-domain typing is possible because the same content can be used independently in different expressions. The types T_1, T_2, T_3 and types T_{10}, T_{11}, T_{12} are from two separate domains. Multi-domain typing makes a simple, single-inheritance hierarchy of semantic types feasible.

Another approach to the combination of different application domains is the blending of concepts [Gog04]. Blending forms a conceptual mapping from a general space to a blend space (the combined meaning) by a conceptual integration of two concepts. This happens via two integration spaces, one for each concept. Clearly, two concepts can be combined in a wide variety of ways. This is illustrated by the common example of “house” and “boat” combined into “houseboat”, “boathouse”, “amphibious vehicle”, or many other concepts [GH04]. Blending however requires a degree of formality of the definitions of concepts that would hurt the ease of use of semantic types.

4.6.2 Openness and Dynamics

As already discussed in section 2.1.4, open and dynamic systems are highly desirable as they enable the systematic support of application domains that are too diverse for static information systems. Openness refers to the way entities are modeled. Under openness users are not restricted to describe entities in a predefined way, but can choose how to most adequately describe the case at hand, hence the model is open. Important application areas of open modeling paradigms are the social sciences, which—due to their inherent subjectivity—cannot work with static descriptions [SBS05], or emerging application domains, which are not yet fully understood and therefore cannot be modeled a priori. To realize the benefits of openness, the supporting system must be dynamic (also see section 2.1.4) in the sense that it can autonomously react to its users' wishes.

Asset Expression are well suited for open modeling of entities. Not only can each expression take the most appropriate explanation path, existing expressions can also be easily extended to augment the explanations to suit particular needs (e.g., those of a new target audience). This section illustrates some use cases of open modeling by Asset Expressions. The next chapter deals with computer systems to support such models.

Building on existing Asset Expressions, expression creators can construct further expressions. Motivations for this are manifold and determine the style of such additions:

- *Expression of a personal point of view.* By adding an abstraction and binding it with another expression information can be added to emphasize a personal point of view on the described entity.

- *Identification of further explanation needs.* The descriptions given in an expression can be insufficient for some users to understand the content. However, these users are often able to point out which part of the content it is that requires additional information. They can record this need for information by adding an abstraction with an appropriate selector to the expression. However, these abstractions are left open without a corresponding application. An expression containing such open abstractions is called an *incomplete description.*

- *Providing further explanations to identified needs.* Experts in the application field, who have sufficient understanding of the content, can then fill the explanation needs expressed in open abstractions by providing applications of further expressoins.

In the last two cases what is called a *partially applied* expression emerges. Unlike, e.g., record- or object-based description mechanisms, the descriptions in Asset Expressions do not have to be captured in one step. Instead, they can be incrementally constructed.

4.7 Grammar

Following is the grammar of the Asset Expression Query Language (AEQL), a subset of which (defined by *constructor*) is the language used to construct Asset Expressions. The grammar is given in Extended Backus Naur Form (EBNF). The syntax used here is that of the World Wide Web Consortium[4]. The syntax is plain text, which means that content cannot be included directly. Instead the content constructor relies on URLs to specify its content representation.

Expressions can be given directly as constructors of variables, applications, abstractions, or content:

constructor ::= *abstraction* | *content* | *parenexp* | *selector*

variable ::= *qualifiedname lift*?

[4] see http://www.w3.org/TR/REC-xml/#sec-notation

abstraction ::= *lambda variable* ":" *typename* "." *expression*
parenexpr ::= "(" (*builtin* | *expression*) ")" (*expression* | *lift*)?
content ::= "@" *url* "@" ":" *typename*
selector ::= "<" *name* "," *address* ">"
address ::= (∼">")*
lift ::= "^ " *typename*
builtin ::= "child" | "expose" | "hole" | "cut" | "match-app" | "match-abs" | "nosubs"

Lifting is treated specially for simple variables to allow their lifting without having to enclose them in parentheses. The rule *builtin* is not strictly necessary as references to these names can be met by supplying appropriate expressions in a library of default expressions. The *constructor* rule covers typed Asset Expressions as described in chapter 3. The query language AEQL also includes constructors but adds the language parts for query tasks such as navigation and filtering. Systems in which users work graphically with a visual representation can generate this text syntax while still allowing their users to view the content in place.

expression ::= *varquery* | *abstraction* | *content* | *parenexp* | *selector* | *traitcreate*
traitcreate ::= "create" *name* " " *expression*
traitdef ::= "trait " *name* (" refines " *name*)? " of " *typename* " with " *absprefix*+
absprefix ::= *lambda variable* ":" *typename* "."
varquery ::= (*variable* | "_") (*pathexpression* | ":=" *expression*)

The *varquery* rule is somewhat too lax, it allows assignments to the default query context. This needs to be resolved in a semantic anlysis. This is also the case for the *parenexpr* rule.

At the first level, query expressions and variable construction are treated together as variables can serve as context for queries. Path expressions filter the set from the context. A path expression is composed of—possibly multiple—steps, each of which has a path and an axis, potentially also a predicate:

pathexpression ::= (*pathstep*)+
pathstep ::= *path axis* (*predicate*)?
axis ::= "*" | "abstraction" | "application" | "variable" | "expressions" | "operand"
| "operator" | "parent"
path ::= *simplepath* | *mainpath* | *allpath*
simplepath ::= "/"
mainpath ::= "//"
allpath ::= "///"

Predicates are expressed as follows:

predicate ::= "[" *orExpr* "]"
orExpr ::= *andExpr* ("or" *andExpr*)?
andExpr ::= *typeExpr* ("and" *typeExpr*)?
typeExpr ::= *comparisonExpr* (":" *typename*)?
comparisonExpr ::= *endExpr* (*comparator endExpr*)?
endExpr ::= (*varquery* | *self* | "(" *orExpr* ")")

Type names are qualified names. A qualified name can be given in three ways: as a fully qualified name consisting of complete namespace and a local name, as an abbreviated fully

qualified name consisting of a namespace prefix and a local name, and as a local name that is valid with respect to the current namespace.

typename ::= *qualifiedname*

qualifiedname ::= ("<" *name* ">" *name* | *name* "#" *name* | *name*)

self ::= "."

Finally, the structures of names and URLs as used above are defined:

digit ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"

letter ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

urlChar ::= *letter* | *digit* | "-" | "_" | "." | " " | "!" | "*" | "'" | "(" | ")" | ";" | ":" | "@" | "&" | "=" | "+" | "$" | "," | "/" | "?" | "%" | "#" | "[" | "]"

name ::= *letter*+

url ::= *name* "://" *urlChar*+

comparator ::= "=" | "in"

lambda ::= "$"

Chapter 5

Systems for Asset Expression Support

Experience has shown that work on large amounts of content has always benefitted from system support [GST00, CKM02]. This is also true for work with Asset Expressions. In fact, besides just managing plain content, systems supporting Asset Expressions need to offer the usual functionality on Asset Expression level: Storage, retrieval, presentation, and querying come to mind. There are also some special needs, arising from specific qualities of Asset Expressions: Deeply structured content requires special system support (as content is no longer opaque). Migration paths to traditional information systems also need to be addressed.

5.1 Scope of Asset Expression Systems

The preceding chapters introduced Asset Expressions which provide flexible means to model real-world entities based on multimedial content. As such content is nowadays mostly available in digital form, its handling requires the support of computerized systems. Hence, to employ the modeling means of Asset Expression on this content, system support for Asset Expressions is required. An information system which is based on Asset Expressions is called an *Asset Expression System* (AES) in the following.

As Asset Expressions are normally created by domain experts, these experts form the primary group of users of an AES. The system must therefore be designed with their needs in mind. Roughly speaking, the scope of an AES is therefore to provide individuals or small groups with the means to use Asset Expressions productively. Likewise, an AES is not built to handle massive amounts of content or to create extremely large networks of such content. Handling a large number of entities from an application domain will benefit greatly from the use of modeling techniques more structured than Asset Expressions [BJJW97, chapter 10]. Much research has been done in the area of highly structured approaches, such as relational databases and information systems based upon conceptual schemata.

As usual in software engineering, requirements to Asset Expression Systems can be broken down into two categories: functional and non-functional ones. The former describe specific behavior of the system, the latter give criteria to evaluate the system's overall operation in ways not pertaining to a specific functionality.

Functional requirements are:

- The ability to create Asset Expressions from scratch by the means introduced in chapter 3 or to modify existing ones. Note that modification of expressions seems to conflict with the reference semantics of Asset Expressions (see chapter 3) as there are no operations available to modify expressions "in place", i.e., without using a reference to the expression.

However, in editing systems it greatly facilitates the general editing of expressions as well as the correction of errors in particular if existing expressions can be modified. Formally, modifications are rebindings of existing names to the modified expressions.

- Otherwise manage the lifecycle of expressions. While again not prescribed by core Asset Expressions, lifecycle management issues, such as deletion of expressions, arise with all realistic usage scenarios. Other lifecycle issues arise when sharing expressions between users to support collaborative work.

- Navigate Expressions. Asset Expressions store information in networks freeing users from an imposed linear structure (as would for example be the case in printed media). AESs must enable users to navigate these networks. Such network-based navigation is similar to hypermedia systems [BVA+97].

- Retrieve defined Asset Expressions. In the simplest form, this can be done by name, but structural as well as content-based retrieval need to be available. Such functionality is central to the AES' role in supporting the work of domain experts. Systems which only provide creation facilities hinder communication among users and make reuse of expressions difficult.

- Act as foundation of higher level services on Asset Expressions. Examples of this will be discussed in chapter 7: Machine-guided semantic typing or construction of intensional conceptual models.

Besides the functional requirements outlined above, there are also non-functional ones. Chief influences on these requirements are the user groups of an AES.

- Flexibility with respect to the way entities are modeled: pass on the flexibility of Asset Expression to users of the system without artificially limiting users in their modeling freedom. Such limitations are imposed by schema-based systems and should specifically be avoided here.

- Helpful visual representation of both multimedial content in general as well as Asset Expressions in particular.

- A medium level of computer literacy must be sufficient to productively use the system. This means that for some concepts of Asset Expressions strong guidances are required in the system to ensure their proper use. Sub-typing is an area typically not well-understood by non-computer-savvy users. Therefore handling of semantic types needs particular attention.

- Low infrastructure requirements. This includes a simple installation mechanism as well as few or loose dependencies on outside systems.

5.2 A General Architecture for Asset Expression Systems

Contemporary approaches to information system creation suggest the generation of such systems from a conceptual model of the application domain. The Model Driven Architecture approach [MM03, Sel03] by the Object Management Group as well as the approach to Conceptual Content Management outlined in section 2.1.4 are examples of this. However, the idea has been proposed—and applied to limited scenarios—much earlier [SWBM89]. Based on the conceptual model of the application domain, a generation facility [Seh04, SBS06, SHZ04, AK03] creates an information system.

A key enabler of system generation is the identification of functionality that is orthogonal to the application domain. This functionality can be codified in the generation facility and

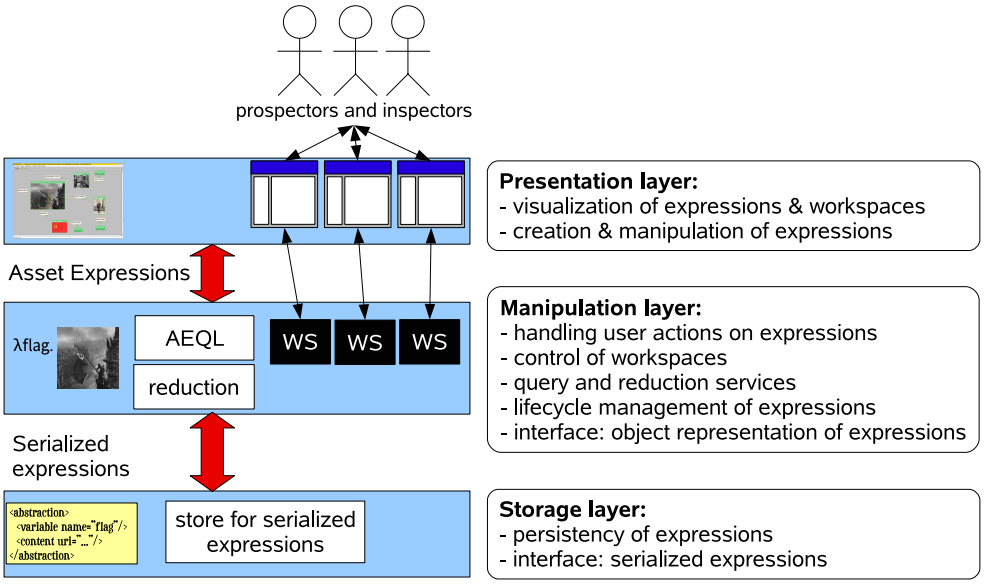


Figure 5.1: *Three-tier architecture for a system based on Asset Expressions*

applied to the application domain each time a system is generated. The set of functionalities to be available in the generated system is determined by configuration of the generation facility such that a system can be generated without loosing the ability to customize its functionality. This is useful if different application domains are to be supported by systems with similar functionality.

Despite these current trends to system generation, AESs are not generated for two reasons. AESs have a fixed functionality which is not driven by the application domain (i.e., the domain for which the users model expressions). Therefore, there is no need to generate a system to meet a particular set of functional requirements. In fact, AESs are highly specialized, domain-specific systems. In general, their domain is the modeling and understanding of some particular application domain. Their purpose thus is *not* to provide the best possible support for work done in this specific application domain. A fishery expert is able to express the domain of fishery by modeling the involved entities. However, the AES does not support the expert in handling a fish trawler. As Asset Expressions are not based on an (open) schema, there also is no need for generation to accommodate schema-dependent parts of the system.

It is therefore sufficient to handle Asset Expressions in a generic system for any application domain. The creation of more static information systems for this domain, which are based on these Asset Expressions, is described in section 7.2.

A general setup for an AES is shown in figure 5.1. It is separated in three layers which fulfill the usual presentation, manipulation, and storage tasks. The *storage layer* is responsible for the persistence of expressions. On the *manipulation layer* various services are provided to create, manipulate, retrieve, etc. Asset Expressions. Finally, the *presentation layer* interacts with users by visualizing Asset Expressions and passing on user input to the manipulation layer below. In the following, the layers are described in more detail from bottom to top.

5.3 Storage Layer

The persistent storage of data is well-understood for a variety of data models, such as relational [LS87, EN94] or semi-structured [Bun97, BPSM+06] ones. To provide persistency services in AESs generic components can be used. While the data model of the persistency

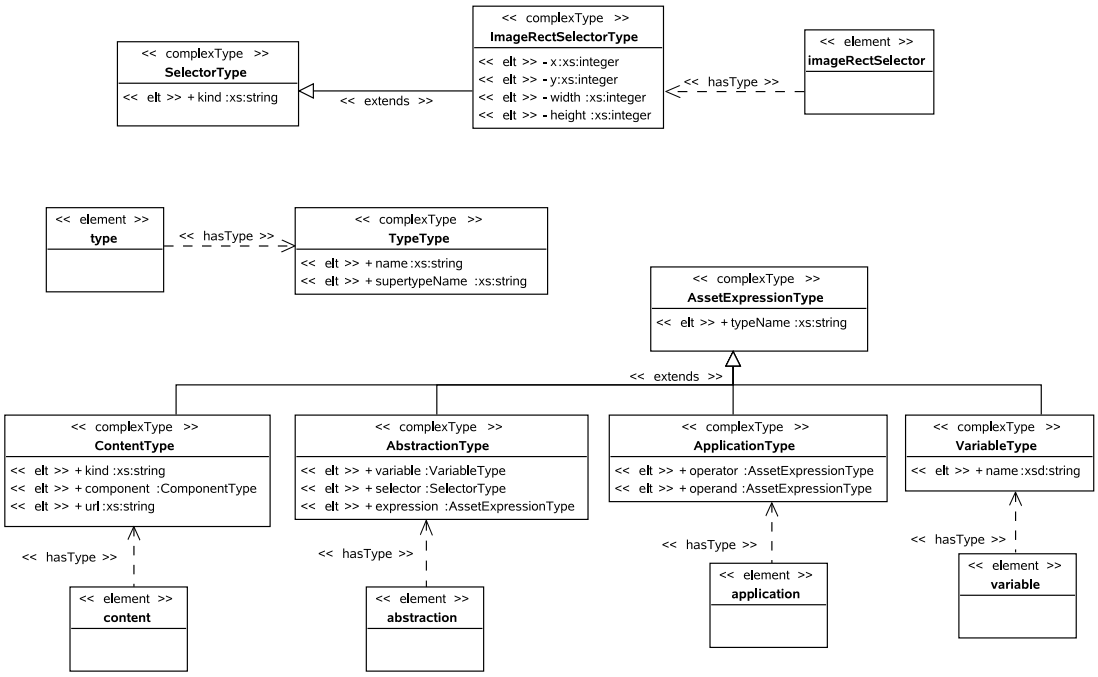


Figure 5.2: *Logical view of the semi-structured data model for the storage layer. The model has been simplified: not all AssetExpressionTypes are shown, the selector model is reduced.*

component does not really matter, one made for semi-structured data suggests itself. The reason for this is the tree structure of Asset Expressions which can conveniently be mapped into semi-structured documents with very little impedance mismatch.

Moreover, if a regular format is used in the documents, later retrieval is also facilitated. It should be noted that the storage layer is schema-based (as opposed to the Asset Expressions on the manipulation layer). However, the schema is not that of the application domain of the expressions created in the system. Rather, it is the schema of the application domain of the storage layer: a meta-model of Asset Expressions. If this schema is made according to a normal form (such as the one suggested in [AL04]), the mapping to the data structure of the manipulation layer is facilitated as it is possible to use automatic tools [MPP00, McL02] to generate this mapping. The storage layer provides an interface in terms of serialized expressions to the manipulation layer on top of it.

As a side-effect of the schema-based workings of the storage layer, expressions which are not well-formed cannot be stored. In fact, it can be required by this storage schema that only expressions that are also well-typed can be put into persistent storage by making a type annotation mandatory on each persistent expression. This serves to acertain that persistent expressions can later be used in all AES use cases.

It is suggested to use an XML [BPSM+06] representation of Asset Expressions as the serialized format. Components to handle the persistency lifecycle are then readily available (e.g., [Mei02]) and the mapping of AEQL to the persistency component's query language (XQuery, [BCF+05]) requires relatively little effort—as will be shown in section 5.4.5—because the semi-structured representation on the storage layer can closely resemble the tree-based object model on the manipulation layer.

5.3.1 Semi-structured Data Model for Asset Expressions

Figure 5.2 shows a portion of a semi-structured model for storing Asset Expressions. The figure uses the logical view proposed in [RBG02], which maps XML elements and types onto UML classes. This allows a concise yet relatively precise representation of the model. Each class is annotated with a stereotype which distinguishes elements from types. The model is in element normal form [AL04]. Therefore, the types only contain elements as denoted by the ≪elt≫ stereotype. The elements are either of a complex type (which is described in the diagram) or of a built-in simple type from the namespace for XML Schema xs. Most complex types are associated with an element which is of that type. Asset Expressions are modeled in semi-structured documents according to this model.

The structure of each type of Asset Expressions is modeled in the extensions of the complex type AssetExpressionType. There is one type each for content, variables, abstractions, applications, name references, and name bindings (the last two are not shown in the diagram). The elements of each type reflect the recursive structure of Asset Expressions. Additional types model selectors (with an extension type for coords-abs selectors, many other selectors are not shown) and hierarchies of semantic types. Note that its semantic type is given on each expression.

The depicted model simplifies several aspects of Asset Expressions in an effort to keep the diagram concise. The subtypes of AssetExpressionType, which model the naming of expressions and the referencing of these names, are left out. Component handling is omitted altogether and selectors are attached directly to abstractions. In the full schema, all three handlings are made explicit. Finally, only a single kind of selector is shown.

References to other expressions might occur in the persisted expressions. To implement these references on the storage layer, referencing mechanisms existing on this layer should be used. This has the advantage that references are transparently available, e.g., to queries. A referencing mechanism for XML is XInclude (see section 2.4.2), which inserts a referring element in the place of the reference. When encountering such a referring element, implementations can act as if the reference target were inserted in this place.

5.4 Manipulation Layer

The core functionality of an AES is realized on the manipulation layer. The layer works on an object representation of Asset Expressions. The functionality comprises operations to control the lifecycle of Asset Expressions as well as facilities that serve to organize expression instances and content kinds. In particular, expressions can be constructed from scratch or by recombination of existing expressions. Querying for expressions is also supported based on the AEQL presented in section 4.2. The manipulation layer provides reduction of expressions for the cases that were discussed in section 4.4. Different kinds of content need to be handled on this layer and the appropriate selectors and means to create and reduce components must be available. To this end, a registry of content kinds is introduced which ties together each kind of content with the applicable selectors. This registry is discussed in section 5.4.2.

It is a key task of the manipulation layer to structure the managed expressions in a way that supports each user in handling his or her expressions and that also supports groups of users in controlling their collaboration. The central paradigm to achieve this structuring is that of *workspaces* which will be introduced in the next section.

5.4.1 Workspaces

The application layer offers an extensible set of workspaces to its users. Each *workspace* has a unique name, contains a number of expressions, and is associated with two roles: its prospector and its inspector. The roles are filled by physical users who can take different roles for different

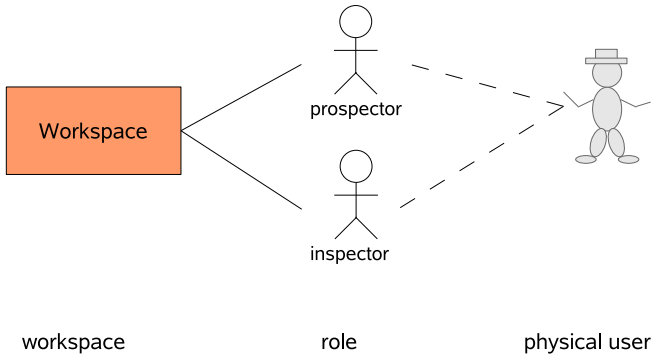


Figure 5.3: *Each workspace has two associated roles—prospector and inspector—which are filled by physical people. While there may well be several inspectors, there will typically only be a limited number of prospectors.*

workspaces. Figure 5.3 illustrates this situation. Often there will be a direct correspondence between workspaces and application domains, but this does not have to be the case and is not enforced by an AES.

The prospector of a workspace is the user who does active work in the workspace by, e.g., creating expressions. Technically, the prospector can be thought of as the one having write access to the workspace. The inspector can only read the expressions but not do any manipulations in the workspace. Students can, for example, pose questions to the teacher by creating expressions which have incomplete explanations to point out parts of content the students does not understand.

Users can be in several roles for different workspaces at the same time. This enables them to create interconnections between the workspaces by referencing expressions in a workspace for that they are inspector from a workspace for that they are prospector:

- A user can be an inspector in several workspaces of other users to use/reference entities created there (creating interrelations with the application domain of the other workspace) or to look at entities presented in the workspace to learn about (the prospector's view of) the application domain.

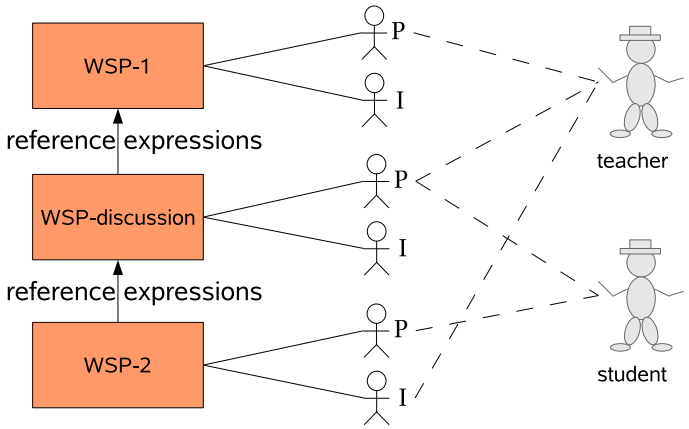


Figure 5.4: *Workspace scenario of teacher-student collaboration*

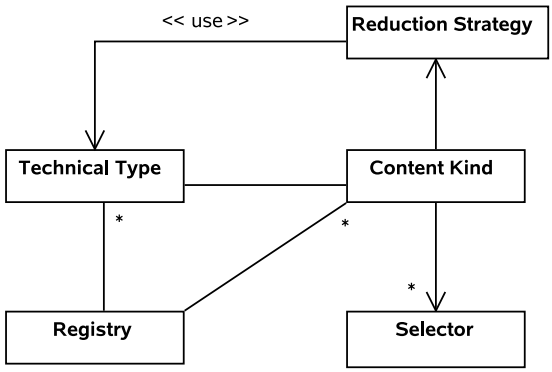


Figure 5.5: *Conceptual model of the registry of content kinds which links technical types of content to the content kinds used in Asset Expressions as well as the corresponding selectors and reduction strategies.*

- A user is prospector of at least one workspace in which this user is the "primary investigator" of the application domain. Prospectors automatically possess investigative capabilities for their workspaces. Therefore a user who is modeled as a prospector is implicitly also inspecting, even though not also modeled as an inspector.

- With symmetric inspector-prospector relations, groups with peer relationships among the members can be founded. Using hierarchical relationships is also possible, for example to model teacher-student relationships as shown in figure 5.4.

The roles of prospector and inspector are also found in the Dockets approach [SS99] as *reviewer* and *evaluator*. Dockets also propose a thrid role of *content provider* which is assumed by the prospector in an AES.

In the context of workspaces it is important that names of expressions can be referred to by a unique name. Each name therefore consists of two parts: a local part and a namespace. This separation is used to ensures uniqueness of names (also compare namespaces in XML [BHLT06]). The local name of expressions is unique in the workspace the expression resides in (the *local* workspace of the expression). Each workspace has a name which is the namespace component of the names of expressions in this workspace. The full name of an expression can then be used to refer to the expression from a workspace other than its local one.

Workspaces provide a point of entry to the presentation layer as they control the scope of the visualization of expressions (compare figure 5.1, more details in section 5.5).

5.4.2 Content Kind Registry

Asset Expressions are not concerned with technical content types (e.g., the encoding of an image or the file format of a textual document) but with an abstraction over the technical types in the form of content kinds as introduced in section 3.2 (also see figure 3.5 on page 54 which shows technical types that are examples of each kind). Implementations of Asset Expression Systems must deal with technical types as well as content kinds because the purpose of an AES is to lift the level of abstraction from opaque content—available in, e.g., a file of a particular technical type—to Asset-Expression-based content of a corresponding kind. Based on these kinds of content the system must be able to apply appropriate selectors to define components.

Figure 5.5 shows a conceptual model of the registry. The registry itself is associated with pairs of (technical type, content kind) which are then equipped with the corresponding selectors

and reduction strategies. For each content kind several selectors might be available, but there is only one reduction strategy.

The AES needs to provide extension points to hook additional pairs of (technical type, content kind) into the system that were not considered during the original development. This is necessary due to the large amount of technical content types available. It is then also possible to fine-tune the set of available selectors to the needs at hand, i.e., to create additional selectors beyond those described in chapter 3.

The reduction strategy is comprised of two parts: implementation of reduction for abstraction based on piece-handling (section 4.3.3) and of reduction for abstraction based on use-handling (section 4.3.2). The former is usually straight-forward as it requires the extraction of a part of the existing content. The latter requires knowledge about other technical content types as it requires substitution of components into larger content. To support this type of reduction, interdependencies between entries in the registry can therefore not be avoided.

5.4.3 Persistency Points

The storage layer provides persistence services to the manipulation layer. It is the task of the manipulation layer to determine when to make use of these services. In particular, this raises the question of when to make expressions persistent. Several possible solutions are available, for example:

- *at once* – any expression will be made persistent directly after its creation
- *explicitly* – users request explicitly that an expression should be made persistent
- *at naming time* – expressions that have names are automatically persistent

When expressions are made persistent at once, it means that the creation of an expression with two explanations over a content will cause five entries in persistent storage: First, the content itself is made persistent, then the subsequent first abstraction is an expression in its own right and therefore made persistent, etc. until finally the second application is created. This causes a high amount of clutter in the form of intermediate expression to be stored with the expression that was actually meant to be created. While this approach achieves a high level of security against data loss, it is deemed overly aggressive.

An explicit approach to persistence requires the user to request persistency for an expression. Document based editors[1] commonly work this way. Experiences show that this request is easily forgotten. Explicit persistency also introduces a transparency issue for the users: Any expression that is reachable (i.e., transitively referenced) from a persistent expression must also be made persistent to ensure that the persistent storage remains in a consistent state. This makes it difficult for users to determine which expressions are in persistent storage.

A working middle ground between immediate and explicit persistency is the persistency of expressions at naming time. In this approach, an expression is made persistent when it is given a name. This avoids the clutter of making all intermediate expressions persistent but also does not introduce an additional feature, which has to be learned and remembered by the user. The idea which determines the granularity of persistency in this approach is that anything the user deems worthy of a name is also worthy to be saved. At the same time, tieing persistency to naming avoids problems with reachability in the persistent storage. Any expression that is reachable from another is so by name. Since the referenced expression has a name, it must also be persistent. Quite conveniently, any persistent expression can easily be retrieved since it is named, making additional "handle" or "expression identifier" mechanisms unnecessary.

Figure 5.6 gives an example of the lifecycle of expressions under the *persistency at naming time* approach. The anonymous expression, which is created first, is not written to persistent

[1] Any editor that has as its unit of abstraction a document of some kind, often also called a "file". If the editor has a visual interface, there usually is a menu titled "File".

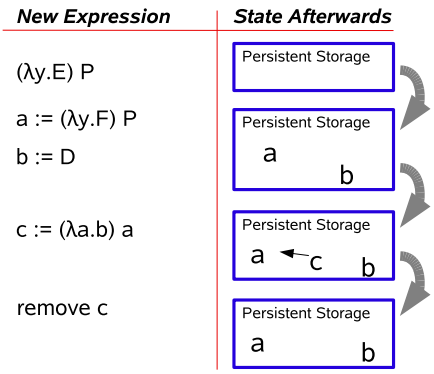


Figure 5.6: *Lifecycle of Asset Expressions in the persistency layer using persistency at naming time.*

storage. The subsequently defined (and named) expressions are. The expression c uses the previously defined expressions and references them by name. When c is made persistent, all prerequisites are already in place.

5.4.4 Reduction

The reduction engine on the manipulation layer follows the substitution rules for the general λ-calculus as from section 2.2.1 as well as the additional specific ones for Asset Expressions described in section 4.4. The implementation of these rules results in a reduction engine which will work on expressions to produce new, reduced expressions, which reside in the manipulation layer. An implementation can be made by directly following the presented rules. The reduction engine needs to work with the content kind registry to retrieve reduction strategies for particular content kinds. This ensures the extensible handling of the diverse landscape of contents, which might be present in an AES.

Besides this reduction on the manipulation layer, there also is a "reduction for the user". In the mind of the user, a full reduction of a presented expression is possible as all necessary context information is available. This context information has been provided by Asset Expressions to fill possible gaps in the user's previous context information (also see page 43). Expressions which have been understood by a user can be reduced by this user because the result of the reduction is representable (it is the understood idea). This cannot happen on the manipulation layer, but some support is possible on the presentation layer as will be discussed below.

5.4.5 Query

The manipulation layer provides services in terms of Asset Expressions to the presentation layer above it by abstracting from the services in terms of semi-structured documents provided by the storage layer below. Included in the services available to the presentation layer is the ability to pose queries in AEQL. These queries need to be evaluated against the data available on the storage layer.

One solution would be to load all documents from the storage layer, to convert them into expressions on the manipulation layer, and to then run the query against these expressions. This approach requires the availability of all expressions on the manipulation layer, which causes an AES to, e.g., load all expressions on startup. The feasibility of this approach is questionable for reasons of scalability.

A different solution is the translation of AEQL into the query language of the storage layer.

The query can then be run on the storage layer and only the results have to be materialized into Asset Expressions. This approach requires, however, a full mapping of AEQL into the query language of the storage layer. One common query language on a semi-structured storage layer is XML Query [BCF+05] (XQuery in the following).

XQuery is a Turing-complete language [Kes02] for semi-structured data in the form of XML documents. Below a mapping of AEQL onto XQuery is provided. Since both XQuery and AEQL work on tree structures and are also somewhat similar in their approaches to navigation as well as filtering, this mapping can be achieved by structural recursion on AEQL. The XQuery fragment corresponding to each element of AEQL will be described below.

Implementation: Mapping to XQuery

Let m : AEQL $\rightarrow$ XQuery be a mapping function which translates an AEQL expression into the equivalent XQuery expression. This function is then defined piece-wise as shown below, but not all XQuery fragments are printed as they can become very lengthy. For the conversion of names a function $name(a)$ exists, which simply returns the fully qualified name of a, being either a semantic type or a name reference. An appropriate XQuery preamble is assumed for the following mapping. It contains all required XML namespace declarations, at least the one for the Asset Expressions schema bound to the prefix `ae` and one of the XInclude schema bound to `xi` as well as implementations of the functions used in the XQuery code created by m.

During the mapping, a context of variables keeps track of transiently defined variables from AEQL **foreach** ... **in** ... expressions.

- **simple path**: This path is mapped to the direct "descendant" XPath-axis:

$$m(/\mathsf{e}) = /m(e)$$

- **main path**: The main path cannot be mapped directly to an XPath axis. Instead, an XQuery function `mainAxis(n as element*)` is used which filters a given sequence of nodes `n` according to the semantics of the main path by starting with the descendant-or-self nodeset and removing from it any node that is only reachable via an `operand` element.

$$m(d//\mathsf{e}) = \texttt{mainAxis}(m(d))m(e)$$

- **all path**: For the all path a direct mapping to the XPath-axis "descendant-or-self" is used:

$$m(///\mathsf{e}) = //m(e)$$

- **type of**: Due to subtyping all subtypes of the given type have to be searched for. This is not as bad as it seems at first glance as the types of nodes in the context do not have to be computed but are given explicitly. An XQuery function which implements the rules for type substitutability from section 3.3 is assumed. Its signature is isSubstitutable(t as xs:string, u as xs:string) as boolean.

$$m(d\colon T) = \texttt{isSubstitutable}(m(d)/@\texttt{typeName},\ name(T))$$

  `@typeName` references the attribute that annotates the type on each stored expression as described in the storage layer schema given in figure 5.2.

- **expression kind axes**: It is necessary to constrain the XML element name, for the /abstraction axis this is:

$$m(/\mathsf{abstraction}\ e) = /\texttt{abstraction}\ m(e)$$

  The other axes are mapped analogously.

- **relationship kind axes**: A similar argument applies to the relationship axes. These can also be mapped to XQuery name test due to the rich model at the storage layer which makes these axes explicit. As an example, consider the operator axis:

$$m(/\texttt{operator}\ e) = /\texttt{operator}\ m(e)$$

  Again, the other relationship axes are mapped analogously.

- **constructors**: The expression constructors for Asset Expression map directly to the respective element constructors in XQuery:

$$\begin{aligned} m(\lambda d.e) = &< \texttt{ae:abstraction} > \\ &\quad m(d) \\ &\quad < \texttt{ae:expression} > m(e) < /\texttt{ae:expression} > \\ &< /\texttt{ae:abstraction} > \\ m((d)e) = &< \texttt{ae:application} > \\ &\quad < \texttt{ae:operator} > m(d) < /\texttt{ae:operator} > \\ &\quad < \texttt{ae:operand} > m(e) < /\texttt{ae:operand} > \\ &< /\texttt{ae:application} > \end{aligned}$$

  and so forth for variables, content, name definitions, and name references.

- **foreach**: Iteration is mapped using the `FLWR` construct in XQuery (see page 41). AEQL variables which are introduced need to be added to the variable context before mapping of the embedded parts. If multiple variables are introduced in **foreach** (compare the examples in section 4.5), multiple XQuery `for` clauses are needed.

$$m(\textbf{foreach}\ v\ \textbf{in}\ e\ \textbf{where}\ p\ \textbf{return}\ r) = \texttt{for}\ m(v)\ \texttt{in}\ m(e)\ \texttt{where}\ m(p)\ \texttt{return}\ m(r)$$

- **name references**: There are two cases of name references: transient variables created by **foreach** ... **in** ... expressions and references to (persistent) Asset Expressions. If the use of the former is detected from the variable context, the name reference is converted into an XQuery variable:

$$m(v) = \$name(v)$$

  In case of a reference to an Asset Expression, the corresponding XInclude constructor is inserted:

$$m(v) = < \texttt{xi:include href} = "\$name(v)" / >$$

Name-references are only transparent to queries if the underlying implementation (XMLDB) can index XIncludes in conjunction with XQuery. If such an implementation is not available, a workaround is the expansion at storage time [eXi] of XIncludes. However, this comes at a high maintenance cost when rebinding names as all previously expanded references have to be found and replaced. The exact cost thus depends on the amount of references used and the cost of sub-tree replacements in XML documents.

5.5 Presentation Layer

Capturing entities through explained multimedial content requires domain experts to provide the appropriate descriptions. Obviously, the available description mechanisms must be appropriate to the domain expert. In particular, complex mechanisms that aim to provide as rich a model as possible often assume familiarity with computer science concepts such as class-based inheritance [DT89, BB04]. Asset Expressions tip the scale of formality [MGMW05] towards

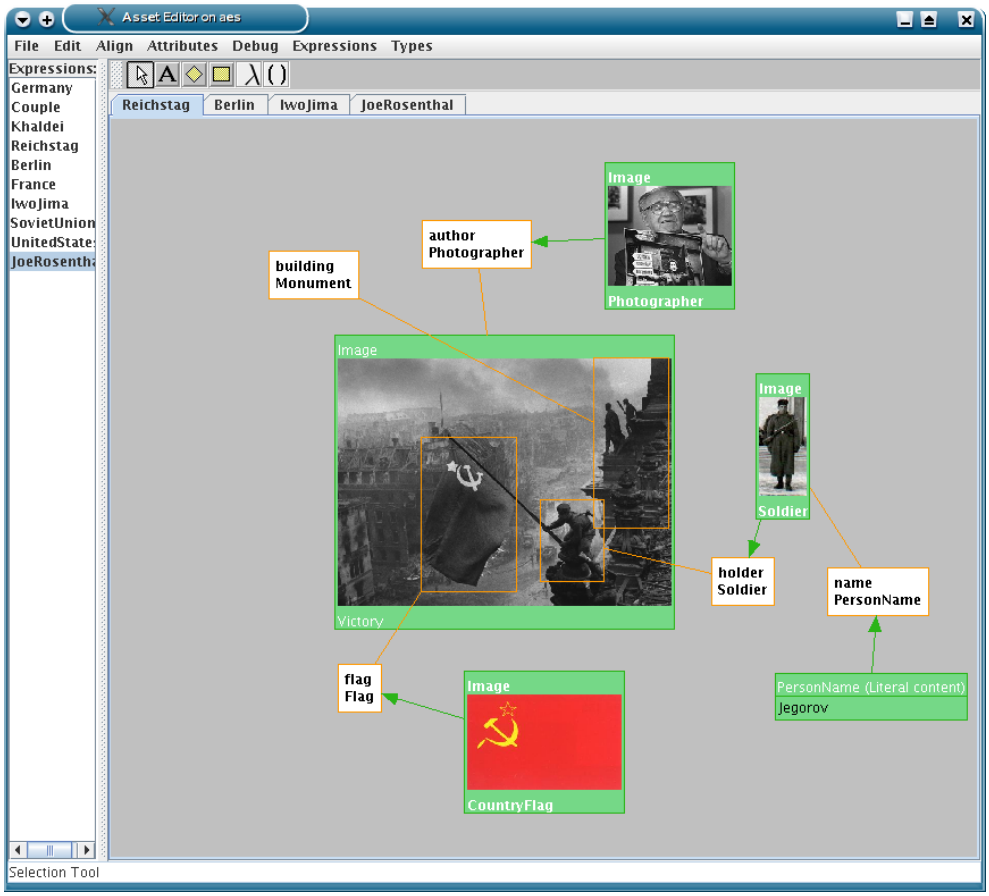


Figure 5.7: *A graphical user interface of manipulating Asset Expressions. The notation used is very similar to that introduced in chapter 3. The "building" abstraction is not filled by an application.*

the user by having only two essential concepts: abstraction and application. Nevertheless, representing these concepts well on the presentation layer remains crucial to the success of AESs.

A central issue of the presentation layer is the adequate presentation of expressions to the user without having available any additional information on the application domain. The presentation layer in an AES is based on the visual notation for Asset Expressions which was introduced in chapter 3. Screenshot of such a presentation layer is shown in figures 5.7 and 5.8. Based on this visual notation domain experts are enabled to create expressions describing the entities at hand without having to fully understand the underlying formalism.

Asset Expression systems take an approach to user interaction described aptly in [BCM99] (variables for back-references removed):

> "In the model of visual HCI [human-computer interaction] we propose, the dialogue between human and computer is modeled by two processes of interpretation-materialization [...]. The two participants, namely the human user and the computer, communicate by materializing and interpreting images visible on the computer screen at successive instants [...]."

The presentation layer displays expressions by breaking them down into their components.

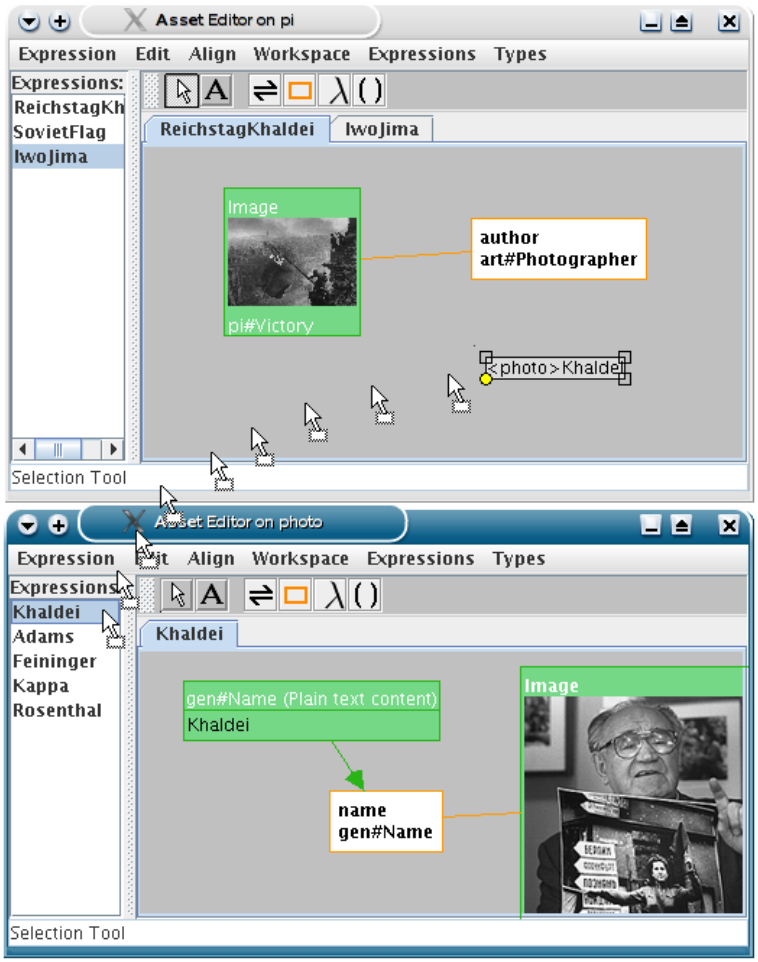


Figure 5.8: *Drag-and-drop between workspaces.*

For each component a visualization is available which might be based (for example in the cases of abstractions and applications) on the visualizations of further subexpression. A similar approach is described in [ROH05] to display RDF[2] documents: To display these documents, recursion along the structure of the documents is used. A distinction is made between global views (which deal with many documents at a time) and local views (which show the details of a document). This gives rise to the questions of presentation granularity, integration of different granularities, as well as navigation through and selection from the repository. A similar distinction is made in the presentation layer of the AES as shown in figure 5.7. On the left is a global view in the form of a list of all expressions in the workspace. In the main part of the window one expression is shown in a detailed local view.

Granularity is also an issue for displaying Asset Expressions. The power of the presentation layer lies in the creation of an appropriate visual co-occurrence of explained content and explaining descriptions. Such a co-occurrence can only be appropriate if the explanations displayed alongside the content are the ones which fill the gaps in the viewer's contextual information. In any case, a full recursive break-down of all expressions which are transitively

[2]RDF: Resource Description Framework [MM04, Bec04] language for expressing machine-understandable descriptions of world-wide-web resources. Also see section 2.3.3.

related to the content does not constitute such an appropriate display. However, what exactly to display, i.e., what granularity to use on the explaining expressions, is up to the particular viewer of the presentation.

Furthermore, a user might not be interested in all details described in the explanations of the expression. The presentation layer provides a tool to hide unwanted explanations to reduce the cognitive load on the user. This tool works by expanding or collapsing explanations at their abstractions (showing or hiding the applied expression).

The components of a content require particular attention in the presentation layer. Depending on the kind of content, different mechanisms need to be available to visualize the components. A visualization of components in an image is shown in figure 5.7. If the extension mechanisms on the manipulation layer are used to add new kinds of content, the presentation layer also requires extension with corresponding display, creation, and editing facilities for components in this content kind. Components can also be used to determine the granularity of explanation display by showing full expansions only if the component is selected. Also see [CCFS95] on the adjustment of detail level in graphs in general and [TH05] on shrinking a large diagram into a "top-k" diagram of its most important nodes.

5.5.1 Navigation

Navigation in Asset Expression Systems can be broken down into two parts. The first part are the actions of the user (e.g., opening a referenced expression) and the means available to express these actions to the system. The second part is the reaction of the system to the user's actions. Here the problem is how to most appropriately cope with the new situation created by the user.

This section is concerned with the actions of users when navigating the expressions in a workspace. There are a number of different kinds of navigation [BBF+02]. When *browsing* (also see [Kwa92]) users pursue two goals: finding the core of interest and searching this core. Browsing ignores the exact structure of an expression, rather users glance over large expressions to find the core of interest. *Directed searching* requires some pre-built structure in the expressions to be searched, which can either be expressed in particular explanations of expressions or dynamically created by appropriate queries (e.g., for all expressions of a certain semantic type).

When navigating expressions, users create a history of expressions they have visited. Many times the navigation does not yield the desired expression—much like clicking a link when surfing the world-wide-web might lead to a document that does not hold the desired content even though the link suggested it. For these dead ends, the system offers the users the opportunity to jump back to a previously visited expression in the history. This form of backwards navigation has been popular with web-browsers for a long time and is also spreading to other types of systems, e.g., integrated development environments.

Hypermedia systems have different possibilities to react to the activation of a link [HBR93]: The existing presentation can be augmented with additional information or replaced entirely. In the model of [HBR93] each link is accompanied by a structured context, which describes the "scope of information which is affected by following a link".

A similar notion can be applied to the presentation of Asset Expressions. However, there is no need to annotate each application with a structured context. Instead, two situations are considered. Each of them is treated with a different context: The expansion or collapsion of applied expressions, and the display of referenced expressions. When expanding or collapsing explanations to produce a co-occurrence of explained and explaining entity which is of appropriate granularity, the existing presentation is augmented with the visualization of the explaining entity. When dereferencing named expressions, the current presentation is replaced by the presentation of the referenced expression.

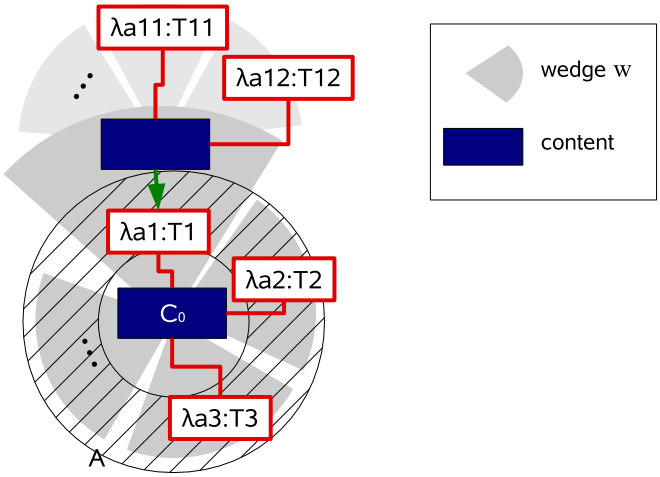


Figure 5.9: *The automatic layout algorithm distributes explanations evenly around the content. This scheme is repeated for sub-expressions.*

5.5.2 Automatic Layout

When users create expressions by using the direct construction tools on the presentation layer, they provide a layout for these expressions at the same time. However, many expressions are not created by these tools, for example the results of those queries that do not simply return available expressions. For such new expressions the presentation layer needs to be able to devise a visualization layout for the expression elements.

Algorithms for automatic layout generation are available in literature, [LF01] provides a survey. Several different methods can be distinguished: simple (e.g., with an explicit layout manager), constraint based, and learning techniques.

This presentation layer uses a simple layout manager, whose layout algorithm is described below. Constraints cannot be used for layout without burdening the user with supplying these additional constraints of each expression which is to be visualized. If this expression has just been created, the amount of work is roughly equivalent to manually laying out the expression.

The layout manager for automatic layout of an expression e works as follows. The algorithm assumes a wedge w which is a portion of a circle defined by start and end angles (the full areas of the wedges are shown in the background of figure 5.9, which illustrates the algorithm).

1. Find the innermost content C_0 by recursively following the /operator path on each application and the /expression path on each abstraction of the expression e until a plain content is reached.

2. Put C_0 in the center of the display area.

3. Determine the set A of all abstractions over C_0.

4. Lay out the abstractions equidistantly on the wedge w around C_0.

5. For each abstraction $a \in A$ compute its corresponding wedge w_a and find the applied expression e_a. Set $e := e_a$ and $w := w_a$ and continue at 1.

The algorithm creates nested wedges on which the expressions are distributed. The wedges become narrower with each level of nesting. Depending on the presentation capabilities of the system, simplifications (e.g., elision of components) should be used on outer levels.

As in other graph editors [PBE98], the presentation layer differentiates between the expression and the expression's layout. The two are stored separately from each other. The encoded layout information is passed on as content to the manipulation layer which in turn puts it into persistent storage.

Chapter 6

Asset Expressions and Conceptual Schemata

Asset Expressions are used to model entities from the real world by creating networks of explained content which describe these entities. Expressions are created on an individual basis and no two expressions have to be alike, neither in structure nor in content. However, in many practical applications a good understanding of the domain of interest leads to opportunities to reuse past effort and thereby arrive at descriptions which are more compact in size as well as exhibit a higher degree of interrelation. Such reuse can, for example, take place by factoring out structural commonalities between expressions or by reusing modeling effort in structurally similar expressions.

Some means in this direction are offered by Asset Expressions. Common descriptions that appear in multiple expressions can be referenced from all occurrences. Expressions can be typed in semantic types, creating groups of expressions that share a common type but can be unrelated in structure. The common type relates them semantically. If some structural interrelation is also desired—for example because it facilitates the collaboration of groups—traits can be used to capture such common structure and serve as templates for future expressions.

Situations might arise where even more structure is desired. In these cases, the high degree of flexibility of modeling with Asset Expressions takes second place to needs which call for a very regular description of application domains. Examples of such cases are information systems which have to support the collaboration of a large number of users—some experts, some novices—or which need to interface with other systems. Both scenarios typically require conceptual models of the application domain to be realized. Practical experience shows that in many such systems (and in fact in all productive CCMSs) classification (i.e., semantic types) and conceptual (as introduced in this chapter) coexist [Nor95].

This chapter describes how a conceptual schema can be obtained from an Asset Expression. To this end, a type system for Asset Expressions is constructed which types expressions with intensionally defined classes. These classes can then be used to construct a conceptual schema of the application domain described by the Asset Expressions, as will be detailed in section 7.2.

If an information system is constructed based on the conceptual schema of the application domain, Asset Expressions can be carried over into this system as they are typeable in the classes the system is based on. Practically, Asset Expressions need to be converted into objects in the software system. This mapping—which will also be presented below—is closely related to the class-based typing rules for expressions. Figure 6.1 illustrates this process. Classes are used to type both expressions and instances. The latter two can—under some conditions—be converted into each other.

The conceptual modeling paradigm used in this chapter is that of Conceptual Content Management (CCM) as described in section 2.1.4. It is a class-based paradigm which describes

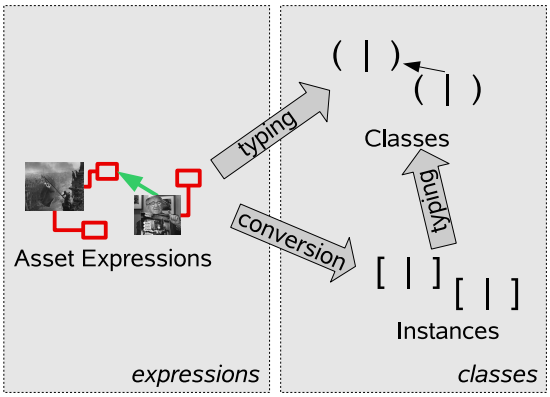


Figure 6.1: *Conversion of freely modeled Asset Expressions into instances of a class-based conceptual schema. The typing of Asset Expressions in classes can fail and therefore determines the expressions that can be converted.*

entities intensionally. Its advantage over other paradigms in the context of Asset Expressions lies in its dualistic modeling of entities based on multimedial content and conceptual descriptions. This matches well with the approach of Asset Expressions to describe entities by explained content. The Conceptual Content Management Systems can be generated from the conceptual schema. This allows open changes to the model and systems that dynamically react to these changes. Openness and dynamics integrate well with Asset Expressions which can be used to support these two properties through the type system presented in this chapter.

6.1 Intensional Typing of Asset Expressions

This section describes a class-based type system AE_C for Asset Expressions. In contrast to the type system $\mathrm{AE}_{\rightarrow}$ of semantic types introduced in section 3.3, the types in AE_C are defined intensionally. They reflect the structure of the descriptions of entities, which is common to many conceptual modeling approaches (e.g., in the Entity-Relationship approach [Che76] or in Conceptual Content Management (CCM) [Seh04]).

Multimedial content plays an important role in Asset Expressions. Matching this, the modeling approach used in the class-based typing is CCM which offers dualistic models of entities through both a multimedial representation and a conceptual description of the entity. These pairs are Asset classes with two compartments: one for content and one of conceptual attributes. This section will use the following notation to denote an Asset class A with medial contents $c_1 \ldots c_p$ and attributes $m_1 \ldots m_k$:

$$A := \left(\begin{array}{c|c} c_1 : H_1 & m_1 : T_1 \\ \vdots & \vdots \\ c_p : H_p & m_k : T_k \end{array} \right)$$

$T_1 \ldots T_k$ are types of attributes, $H_1 \ldots H_p$ are content handle types as defined in section 2.1.3. In this model, the members of the class are ordered. The order of members of the Asset classes defined in CCM is not specified, but the typing rules for Asset Expressions can be stated in a more concise manner if an ordering is assumed.

Also for the purposes of defining the typing rules, an extension operator $\oplus$ on Asset classes

is defined which creates a new Asset class from two existing classes by appending the members of the second to the first:

$$\underbrace{\left(\begin{array}{c|c} c_1 : H_1 & m_1 : T_1 \\ \vdots & \vdots \\ c_p : H_p & m_k : T_k \end{array}\right)}_{=:\mathrm{A}} \oplus \underbrace{\left(\begin{array}{c|c} d_1 : I_1 & n_1 : U_1 \\ \vdots & \vdots \\ d_q : I_q & n_j : U_j \end{array}\right)}_{=:\mathrm{B}} =: \underbrace{\left(\begin{array}{c|c} c_1 : H_1 & m_1 : T_1 \\ \vdots & \vdots \\ c_p : H_p & m_k : T_k \\ d_1 : I_1 & n_1 : U_1 \\ \vdots & \vdots \\ d_q : I_q & n_j : U_j \end{array}\right)}_{=:\mathrm{C}}$$

Each class in this model is assumed to have a unique name. The name is of no further importance in AE_C as this type system is purely structural. The structural nature of the type systen does, however, *not* imply that structural subtyping is used, i.e., with A, B, and C as above it is not the case that $C <: A$ or $C <: B$.

Using this model, the system AE_C of intensionally motivated types will be developed below. Essentially, abstractions are reflected as conceptual attributes and content from Asset Expressions is modeled in the content compartment of the Asset class. The model works on top of functionally-typed Asset Expressions. This is necessary as otherwise no distiction could be made between the types of typed, but otherwise not explained, contents, see the typing rules for content in section 6.1.2. These contents—which are the atoms that are assumed to be universally understood without explanation—occur quite often and can be of a variety of semantic types. It would therefore be inappropriate to model them all with the same intensional type in the class-based type system. AE_C is a Church-style system [Pie02, 9.6] in which only semantics of well-typed terms are defined.

AE_C indicates how to convert the expressions into instances of the class they have been typed as. In the following examples these instance are written with square brackets. Intuitively, explanations in the expressions are converted into members in the Asset instance. In the conversion directly applied content (i.e., content that does not require explanations of its own) is mapped onto characteristics in the Asset instance:

$$(\lambda y.(\lambda x.C_1)A)B \quad \Leftrightarrow \quad \left[\begin{array}{c|c} C_1 & x = A \\ & y = B \end{array}\right]$$

If A and B are plain content. If, however, the operand of an application is richly modeled in itself, the operand is converted to a separate Asset instance. The instance for the overall expression then references the instance for the operand:

$$(\lambda y.(\lambda x.C_1)A)(\lambda a.C_2)D \quad \Leftrightarrow \quad \left[\begin{array}{c|c} C_1 & x = A \\ & y = \bullet \end{array}\right] \qquad \left[\begin{array}{c|c} C_2 & a = D \end{array}\right]$$

Such conversions will be discussed in more detail in section 6.2. As they are based on AE_C, some details on this type system follow first.

In the following $\mathcal{S}$ is the set of semantic types used for typing in $\mathrm{AE}_{\rightarrow}$. In this chapter only, the semantic type T of an Asset Expression e will be written as $e \diamond \mathsf{T}$, the colon used previously is used for class-based types here. In the following, AE is the set of all Asset Expressions, A is the set of all Asset instances. $C \subseteq AE$ is the set of all Asset Expressions which are plain content.

6.1.1 Types

The set of ground types for AE_C is slightly more complex than in $\mathrm{AE}_{\rightarrow}$. It is $\mathcal{G} = \mathcal{G}_0 \cup \mathcal{G}_C \cup \{Asset\}$ with the following elements:

- $Asset := (\ |\ )$ the built-in, empty base-class. It will become the super-class of all classes used to type Asset Expressions. *Asset* is an Asset class and thus different from the semantic type Any.

- $\mathcal{G}_0 = \{int, String, Date, ...\}$ the set of base types for characteristics. The types, that can be used for characteristics in Conceptual Content Management, stem from the base language used in CCM systems.

  Mappings of specified semantic types to these characteristic types for classes will be defined in section 6.1.3. A function $map_S(\mathsf{T_S}) : T_0$ is available that implements this mapping. It returns the characteristic type $T_0 \in \mathcal{G}_0$ corresponding to the semantic type $\mathsf{T_S}$ if a mapping is available, *Asset* otherwise. Note that the characteristic types are not sub-types of *Asset*:

  $$map_S(T_S) = \begin{cases} c \in \mathcal{G}_0 & \text{if a mapping exists} \\ Asset & \text{else if } T_S \text{ is a simple semantic type} \\ Asset* & \text{else } T_S \text{ is a list type} \end{cases}$$

- $\mathcal{G}_C = \{Image, Text, ...\}$ the set of handle types for content. Similar to the types in $\mathcal{G}_0$, a mapping to find the handle type in the target system must be supplied. This mapping is implemented by a function $map_C(C) : T_H$ that returns the content handle type $T_H \in \mathcal{G}_C$ depending on the content kind of content C.

The inverse counterparts of both mapping functions are also needed. They are not necessary for typing expressions as in this case there only is a conversion of semantic types/content kinds into Asset types. However, as expressions are also to be converted into Asset instances and vice versa, the inverses of the mapping functions will be needed in section 6.2 where the conversion of instances is defined.

The elements of the set $\mathcal{G}$ as well as more specialized Asset classes can be obtained in three ways:

1. By invoking the functions map_C and map_S for content and characteristic types, respectively.

2. By extending existing class types with the $\oplus$ extension operator. Chains of such extensions always start from *Asset*.

3. By using the type constructors $*$ and $\rightarrow$ on a type A. The former creates a set type $A*$, the latter adds a functional marker to the type: $A \rightarrow open$. Functionally marked types are legal in intermediate typings only. Their occurrance in the final type for an expression indicates that this expression is ill-typed. Use of the functional marker will be explained in detail with the typing rules in the next section.

6.1.2 Typing Rules

This section introduces the typing rules for AE_C. Expressions must be well-typed with respect to $\mathrm{AE}_{\rightarrow}$ as otherwise the mapping functions for content handle types and characteristic types cannot be used.

In this chapter only, the semantic type T of an Asset Expression e is written as $e \diamond \mathsf{T}$, the colon used previously is used for class-based types here. Thus, the context Γ takes the following form in this section:

$$\Gamma ::= \epsilon \mid \Gamma, x \diamond \mathsf{T} \mid \Gamma, C \diamond \mathsf{T} \qquad \text{for variables } x \text{ and contents } C$$

Where ϵ is the empty context. Apart from this, typing rules are stated in the same syntax that was used in section 3.3.4 to describe $\mathrm{AE}_{\rightarrow}$.

Variables are typed in semantic types, their AE_C type can be determined with the mapping function map_S.

$$\frac{}{x_1 \diamond \mathsf{A_1}, \ldots, x_m \diamond \mathsf{A_m} \vdash x_i : map_S(\mathsf{A_i})} \qquad i \in [1, m],\ \mathsf{A_i} \in \mathcal{S}$$

Plain content is typed in Asset classes if its semantic type does not map to a characteristic type. If it does, it is typed with the assigned characteristic type from $\mathcal{G}_0$.

$$\frac{map_S(\mathsf{A}) \in \mathcal{G} \setminus \mathcal{G}_0}{\Gamma \vdash c \diamond \mathsf{A} : (\ c : map_C(c) \mid\ )} \qquad \frac{map_S(\mathsf{A}) \in \mathcal{G}_0}{\Gamma \vdash c \diamond \mathsf{A} : map_S(\mathsf{A})} \qquad c \in C$$

Each content in an Asset class has a name. Since Asset Expressions do not give names for content, the constant name "c" is used. Lists of content are typed with an Asset class that has multiple entries in the content compartment:

$$\frac{map_S(\mathsf{A_i}) \in \mathcal{G} \setminus \mathcal{G}_0, i \in \{1 \ldots N\}}{\Gamma \vdash \{c_1 \diamond \mathsf{A_1}, \ldots, c_N \diamond \mathsf{A_N}\} : \left(\begin{array}{l|} c_1 : map_C(c_1) & \\ \vdots & \\ c_N : map_C(c_N) & \end{array} \right)} \qquad c_i \in C$$

Similarily to the single content case, names are invented for each content.

To achieve intensional typing, abstractions of Asset Expressions are reflected as attributes in the Asset class. Therefore, the class is extended with an additional attribute for each abstraction in the expression. However, as only expressions with complete explanations are well-typed, the new attribute is not given the type that corresponds to the abstraction right away. Instead, a functional type is used to mark that no application has been encountered for this abstraction. If no such application exists in the expression, the attribute in the class will remain of function type. The expression can then be recognized as ill-typed.

The reason for the ill-typedness of expressions with incomplete explanations is as follows. In class-based systems the counterpart of Asset Expressions which model individual entities are instances. These instances are created from classes in *one* step (compare, e.g., the "new" operator in [AC96]). During this step values are supplied to all attributes. These values can be default or null values, but a value has to be given for each attribute. Therefore, Asset Expressions whose class contains function-typed attributes are considered ill-typed as there is no value for the attribute. To change this behavior, one can simply not consider the indicator.

The typing rule for abstractions extends a class B by an additional attribute x which is appended to the existing attributes of B:

$$\frac{\Gamma, x \diamond \mathsf{A} \vdash e : B \qquad B \notin \mathcal{G}_0}{\Gamma \vdash \lambda x \diamond \mathsf{A}.e : B \oplus (\ \mid x : map_S(\mathsf{A}) \rightarrow \mathit{open}\ )}$$

The name of the new attribute is taken from the name of the abstraction variable. *open* is a built-in type flag that signals an open abstraction. The indicator type for the attribute is usually $\mathit{Asset} \rightarrow \mathit{open}$ if A cannot be mapped to a characteristic type. The indicator will be removed by the corresponding typing rule applications. It is important for the interaction of the two rules for abstraction and application that the attributes of the class are ordered.

Attributes of function type (i.e., attributes for which the matching abstraction has not yet been encountered) form a stack at the end of the attribute list. The typing rule for applications moves the attribute to the front of the list when its corresponding application is encountered. It is thus ensured that abstractions and applications will be paired correctly. Multiple abstractions can be nested and are matched with their applications.

Consider the expression $\mathsf{book}_{\mathsf{part}} := \lambda\mathsf{title}\diamond\ \mathsf{Name.C}$ as an example and assume that $\mathsf{C} : (\ |x : T\ )$. Then $\mathsf{book}_{\mathsf{part}}$ is typed in AE_C as follows:

$$\mathsf{book}_{\mathsf{part}} : \left(\ \middle|\ \begin{array}{l} x : T \\ \mathit{title} : \mathit{map}_S(\mathsf{Name}) \to \mathit{open} \end{array}\ \right)$$

Abstractions extend the Asset class by a member that is typed in a function type. Applications collapse these function types of an attribute and prepend the attribute to the class. An ordinary Asset class emerges for well-typed expressions:

$$\frac{\Gamma \vdash e : B \oplus (\ |\ x : C \to \mathit{open}\ ) \qquad \Gamma \vdash f : A \qquad A <: C}{\Gamma \vdash (e)f : (\ |\ x : A\ ) \oplus B}$$

In the context of the application, the actual type of the applied expression is known and can be used as the type for the attribute of the class. The expression

$\mathsf{book}_{\mathsf{full}} := (\mathsf{book}_{\mathsf{part}})\mathsf{SomeName}{=}(\lambda\mathsf{title}\diamond\ \mathsf{Name.C})\mathsf{SomeName}$

is typed as follows:

$$\mathsf{book}_{\mathsf{full}} : \left(\ \middle|\ \begin{array}{l} \mathit{title} : \mathit{map}_S(\mathsf{Name}) \\ x : T \end{array}\ \right)$$

All types generated by explanations are direct subtypes of *Asset*. Characteristic types are defined explicitly and can therefore participate in more detailed inheritance hierarchies. Substitutability of Asset Expressions is defined based on semantic types. It can therefore not be reproduced in intensional types because without loss of generality no injective mapping from semantic types to intensional types can be assumed.

Lifting of expressions changes their semantic type to a more specific one. This has no effect on the intensional structure of the type:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e \uparrow \mathsf{T} : A} \qquad \mathsf{T} \in \mathcal{S}$$

Lists of expressions are typed similarly as before, of course now in AE_C types:

$$\frac{\Gamma \vdash e_i : A}{\Gamma \vdash \{e_1, \ldots, e_n\} : A*} \qquad i \in [1, n], A \in \mathcal{G}$$

All typing rules are summarized in figure 6.2.

6.1.3 Types for Characteristics and Content

Semantic types of Asset Expressions are not preserved in AE_C as semantic types have no defined structure to make them suitable starting points for creating classes. However, two cases have been identified above in which special care has to be taken to determine the type of an expression in AE_C. One such case is the typing of characteristics as part of the concept. Semantic types need to be converted to a specific primitive type for the characteristic. The other case occurs because content is of content handle type—which is technically motivated—in Asset classes, while the semantic type of content in Asset Expressions has no relation to the content's technical type. Therefore, semantic types cannot be converted into content handle types. Instead the content handle type is determined from the content kind.

$$\text{Content}_1 \quad \frac{map_S(\mathsf{A}) \in \mathcal{G} \setminus \mathcal{G}_0}{\Gamma \vdash c \diamond \mathsf{A} : \left(\; c : map_C(c) \;\middle|\; \right)} \qquad c \in C \qquad (6.1a)$$

$$\text{Content}_2 \quad \frac{map_S(\mathsf{A}) \in \mathcal{G}_0}{\Gamma \vdash c \diamond \mathsf{A} : map_S(\mathsf{A})} \qquad c \in C \qquad (6.1b)$$

$$\text{Content}_3 \quad \frac{map_S(\mathsf{A_i}) \in \mathcal{G} \setminus \mathcal{G}_0, i \in \{1 \ldots N\}}{\Gamma \vdash \{c_1 \diamond \mathsf{A_1}, \ldots, c_N \diamond \mathsf{A_N}\} : \left(\begin{array}{l} c_1 : map_C(c_1) \\ \vdots \\ c_N : map_C(c_N) \end{array} \middle| \; \right)} \qquad c_i \in C \qquad (6.1c)$$

$$\text{Abstraction} \quad \frac{\Gamma, x \diamond \mathsf{A} \vdash e : B \qquad B \notin \mathcal{G}_0}{\Gamma \vdash \lambda x \diamond \mathsf{A}.e : B \oplus \left(\; \middle| \; x : map_S(\mathsf{A}) \to \mathit{open} \; \right)} \qquad (6.1d)$$

$$\text{Application} \quad \frac{\Gamma \vdash e : B \oplus \left(\; \middle| \; x : C \to \mathit{open} \; \right) \qquad f : A \qquad A <: C}{\Gamma \vdash (e)f : \left(\; \middle| \; x : A \; \right) \oplus B} \qquad (6.1e)$$

$$\text{Lifting} \quad \frac{\Gamma \vdash e : A}{\Gamma \vdash e \uparrow \mathsf{T} : A} \qquad \mathsf{T} \in \mathcal{S} \qquad (6.1f)$$

$$\text{Lists} \quad \frac{\Gamma \vdash e_i : A}{\Gamma \vdash \{e_1, \ldots, e_n\} : A*} \qquad A \in \mathcal{G} \qquad (6.1g)$$

Figure 6.2: *Intensional typing of Asset Expressions*

To obtain types for characteristics from the semantic type of the corresponding abstraction variable, the function $map_S : \mathcal{S} \to \mathcal{G}$ has been introduced. Asset classes use types from their base language (which is also the language a CCMS is implemented in, see section 2.1.4) as types for the characteristics. With the set of these types $\mathcal{G}_0$ and the set of semantic types $\mathcal{S}$ a mapping $M_{C-S} \subseteq \mathcal{G}_0 \times \mathcal{S}$ can be defined which provides pairwise translations. The function map_S can then be defined more concretely as:

$$map_S(T_S) := \begin{cases} T_C & \text{if } \exists p \in M_{C-S} \mid p = \{T_C, T_S\} \\ \mathit{Asset} & \text{else if } T_S \text{ is a simple semantic type} \\ C* & \text{else if } T_S \text{ is a list type } T'_S * \text{ and } C = map_S(T'_S) \\ \text{error} & \text{else} \end{cases}$$

It follows from this definition that the elements of the mapping set must be unique with respect to the semantic type.

For content handle types no distinction of list or individual types is necessary. The function $map_C : \mathcal{K} \to \mathcal{G}_C$ can therefore be based directly on a set of mappings $N_{H-K} \subseteq \mathcal{G}_C \times \mathcal{K}$ from content handle types to content kinds:

$$map_C(K) := \begin{cases} T_H & \text{if } \exists p \in N_{H-K} \mid p = \{T_H, K\} \\ \text{error} & \text{else} \end{cases}$$

The inverse mappings $map_S^{-1}(T_C)$ and $map_C^{-1}(H)$ are defined analogously. The first maps Asset classes to semantic types, the second assigns a content kind for use in Asset Expressions to each content handle type used in Asset classes.

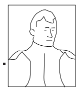
$$map_S^{-1}(T_C) := \begin{cases} T_S & \text{if } \exists p \in M_{S-C} \mid p = \{T_S, T_C\} \\ C* & \text{else if } T_C \text{ is a list type } T'_C * \text{ and } C = map_S^{-1}(T'_C) \\ \text{error} & \text{else} \end{cases}$$

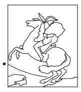
In the inverse, semantic types need to be assigned to all used Asset classes $\mathcal{A}$ and characteristic types $\mathcal{G}_0$ in a set $M_{S-C} \subseteq \mathcal{S} \times (\mathcal{A} \cap \mathcal{G}_0)$. The inverse for content kinds is defined similarly as map_C above:

$$map_C^{-1}(T_H) := \begin{cases} K & \text{if } \exists p \in N_{K-H} \mid p = \{K, T_H\} \\ \text{error} & \text{else} \end{cases}$$

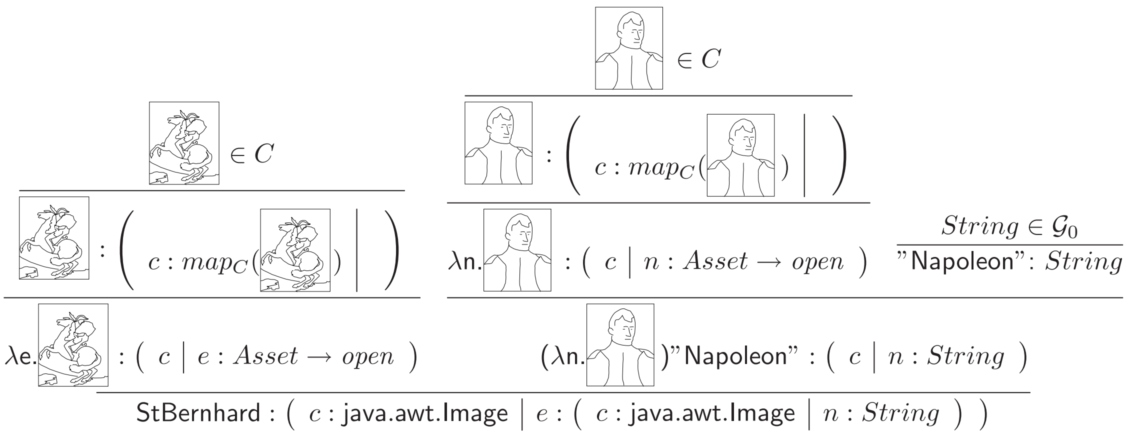
6.1.4 Typing Example

Consider the following Asset Expressions from the domain of political iconography as an example:

Napoleon := (λn: Name.  : Ruler)"Napoleon": Name

StBernhard := (λe: Ruler.  : EquestrianStatue)Napoleon

In AE_C the expression StBernhard is typed as if the referenced expression were inlined. The expression StBernhard can be typed in Asset classes with the following type inferrences. The types of the contents c (java.awt.Image in all cases) have been omitted for brevity everywhere but in the last conclusion.



In terms of Asset classes the type of the operands of applications cannot be checked against a declaration of the corresponding abstraction variable as those variables are typed in semantic types. Rather, the Asset class of the concept attributes e and n, which model the explanations, are derived based on the structure of the operand.

6.2 Converting Asset Expressions to Asset Instances

Given the type system AE_C which types expressions in Asset classes, converting Asset Expressions into instances of Asset classes becomes interesting. Once a class-based conceptual schema is available for Asset Expressions, this schema can be used in many tasks, information integration [RB01, BSG+04] and system generation [SBS06] come to mind as examples. This section describes a conversion function which is able to translate Asset Expressions that are well-typed according to AE_C into Asset instances. The conversion is defined piece-wise on Asset Expressions in a manner rather similar to the typing rules of AE_C.

Asset instances are written with square brackets to differentiate them from Asset classes. Instances are also divided into two compartments: content and concept. Both compartments contain concrete bindings for their members. The combination $\oplus$ of two Asset instances is defined analogously to that of Asset classes as follows:

$$\left[\begin{array}{c|c} c_1 & m_1 : T_1 = v_1 \\ \vdots & \vdots \\ c_p & m_k : T_k = v_k \end{array}\right] \oplus \left[\begin{array}{c|c} d_1 & n_1 : U_1 = w_1 \\ \vdots & \vdots \\ d_q & n_l : U_l = w_l \end{array}\right] =: \left[\begin{array}{c|c} c_1 & m_1 : T_1 = v_1 \\ \vdots & \vdots \\ c_p & m_k : T_k = v_k \\ d_1 & n_1 : U_1 = w_1 \\ \vdots & \vdots \\ d_q & n_l : U_l = w_l \end{array}\right]$$

Concept members m, n have types T, U and values v, w. The type is defined in the corresponding Asset class. It is therefore optional to repeat the type in the instance.

6.2.1 Translation Rules

The function $\text{conv}_{\text{AE}-\text{A}}$: $AE \to A$ converts Asset Expressions into Asset instances. The types of the attributes and the content handle types have already been established during typing and thus do not need to be considered during the translation.

The translation is defined piece-wise along the structure of Asset Expressions. Single contents and characteristic values are converted as follows:

$$\text{conv}_{\text{AE}-\text{A}}(c) = \left[\begin{array}{c|c} c & \end{array}\right] \qquad \text{for a piece of content } c \in C$$

and

$$\text{conv}_{\text{A}-\text{AE}}(v : T) = v\text{: } maps_S(T) \qquad \text{for a characteristic value } v$$

Analogously, a list of plain content is translated into an Asset instance with multiple members in the content compartment:

$$\text{conv}_{\text{AE}-\text{A}}(\{c_1, \ldots, c_N\}) = \left[\begin{array}{c|c} c_1 & \\ \vdots & \\ c_N & \end{array}\right] \qquad \text{for pieces of content } c_1 \ldots c_N \in C$$

These three rules require that any content that is representable in Asset Expressions can also be represented in Asset instances. If this is not the case, the corresponding type mapping should not have been defined. The type of the expression determines which rule is used as defined in section 6.1.3. The abstraction and application are translated in one step as the Asset instance requires a value of the attribute.

$$\text{conv}_{\text{AE}-\text{A}}((\lambda m_1.e_1)e_2) = \left[\begin{array}{c|c} \cdot & m_1 = \text{conv}_{\text{AE}-\text{A}}(e_2) \end{array}\right] \oplus \text{conv}_{\text{AE}-\text{A}}(e_1), e_1 \in AE \qquad (6.2)$$

The name of the abstraction variable is used as the name of the member. For this rule it does not matter whether the operand is also an Asset instance or a characteristic value.

Lists of expressions are converted into lists of Asset instances by converting each expression individually and adding it to the list:

$$\text{conv}_{\text{AE}-\text{A}}(\{e_1, \ldots, e_N\}) = \{\text{conv}_{\text{AE}-\text{A}}(e_1), \ldots, \text{conv}_{\text{AE}-\text{A}}(e_N)\}$$

Lifting has no effect on the structure (see its typing rule), leading to a trivial conversion:

$$\text{conv}_{\text{AE}-\text{A}}(e \uparrow \mathsf{A}) = \text{conv}_{\text{AE}-\text{A}}(e)$$

Analogously to the conversion from Asset Expression to Asset instances, the reverse direction is also defined piece-wise on structure, but on that of Asset instances. In this direction the constructed Asset Expressions need to be equipped with semantic types as specified in the inverse mappings for Asset classes. The function $\text{conv}_{\text{A}-\text{AE}}$: $A \rightarrow AE$ is defined below.

In general, a roundtrip of conversions $\text{conv}_{\text{A}-\text{AE}}(\text{conv}_{\text{AE}-\text{A}}(e_1)) \neq e_1$ will not result in the original expression. With regard to types this is due to the lossyness of the type mapping because several semantic types can be mapped onto a single type in AE_C. Content components can also not be represented in Asset classes or instances and are therefore lost during translation. Nevertheless, an Asset Expression which contains components is well-typed with regard to the AE_C and can also be converted into an Asset instance. A possible remedy for the loss of content components is discussed in the next section.

Instances which only contain content are converted to this content. An attempt to find a semantic type based on the content handle type is made but it is expected that additional liftings to more specific semantic types are necessary as there will rarely be a one-to-one mapping of handle types to semantic types. Asset instances can use several contents:

$$\text{conv}_{\text{A}-\text{AE}}(\left[\begin{array}{c|c} c_1 & \\ \vdots & \\ c_N & \end{array}\right]) = \{c_1\text{: } map_C^{-1}(c_1), \ldots, c_N\text{: } map_C^{-1}(c_N)\}, \quad N > 1$$

or just a single content:

$$\text{conv}_{\text{A}-\text{AE}}(\left[\begin{array}{c|c} c & \end{array}\right]) = c\text{: } map_C^{-1}(c)$$

Concept attributes from Asset instances are converted to pairs of abstraction and application to model the attribute as well as the bound value. The type of the abstraction variable is obtained from the inverse mapping of semantic types. To ensure that the applied expression is of matching type, a lifting is used.

$$\text{conv}_{\text{A}-\text{AE}}(\left[\begin{array}{c|l} c_1 & m_1 = v_1 \\ \vdots & \vdots \\ c_p & m_k : T_k = v_k \end{array}\right]) =$$

$$\left(\lambda m_k\text{: } map_S^{-1}(T_k).\text{conv}_{\text{A}-\text{AE}}(\left[\begin{array}{c|l} c_1 & m_1 = v_1 \\ \vdots & \vdots \\ c_p & m_{k-1} = v_{k-1} \end{array}\right])\right)\left(\text{conv}_{\text{A}-\text{AE}}(v_k) \uparrow map_S^{-1}(T_k)\right), k \geq 1$$

Values of attributes are represented as content in Asset Expressions and simply copied over. The semantic type T of the attribute is converted through the inverse mapping for semantic types $map_S^{-1}(T)$.

6.2.2 Conversion of Content Components

Object-oriented type systems usually do not support multimedial content at all. While Assets offer some support by explicitly accomodating content alongside its conceptual description, they have no built-in notion of content components. However, as such components are an important means to describe and richly connect entities in Asset Expressions, their availability is highly desirable for a pratical conversion of Asset Expressions into Asset instances.

Conceptual Content Management offers means to combine models for different application domains. This facility can be used to provide a meta-model of content components and to combine it with the general Asset model. In this approach, content components are not made part of the core model but added as a separate domain, see figure 6.3. The Asset class Component allows content components to be defined based on a content in an Asset. This components

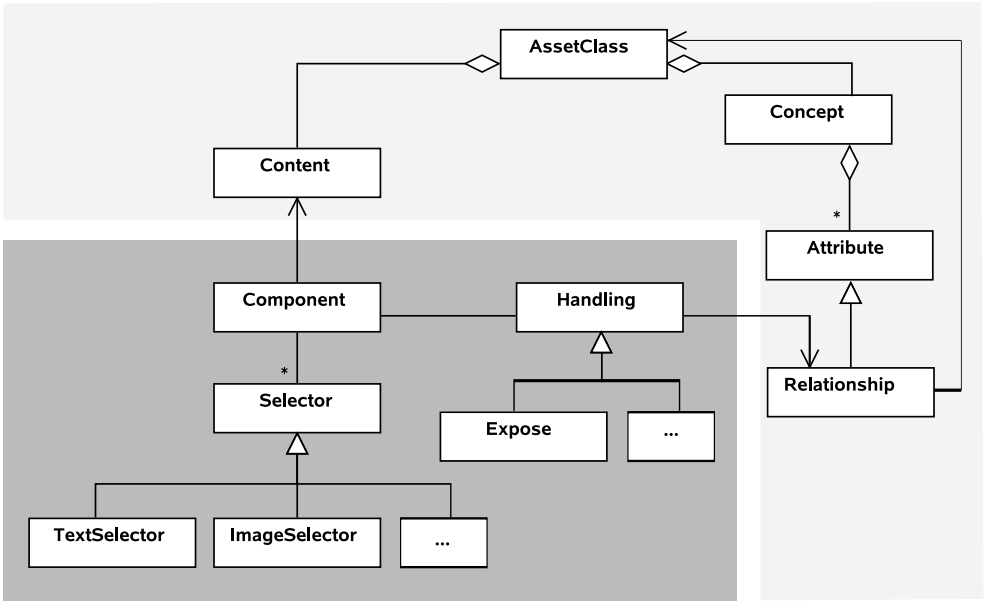


Figure 6.3: *An Asset model for content components. The model for the Asset domain is shown with light background, the one for content components with dark background.*

uses—just as content components in Asset Expressions—Selectors to provide addressing. Different types of selectors are available to accomodate various kinds of content. The defined components are based on the Asset instance whose content they use. This instance is called the *base Asset*. Through a Handling (see section 4.3 on different handlings for components) the component is connected to an Attribute of the base Asset which references another Asset instance (the operand of the application in Asset-Expression terms).

Using this model, content components can be converted into explicit Asset instances alongside the instances describing the entities modeled in the converted Asset Expressions. Accordingly, Conceptual Content Management Systems which are generated from such a combination of models require new content components to be introduced using the system's usual means, e.g., pages with forms in a web-based system. Superimposing a component's addressing onto its content—as prescribed in the visual notation of Asset Expressions—is generally not possible in such a system. Dedicated user interfaces based on the Asset model of content components can be built to remedy this situation [Uri05].

6.2.3 Convertible Expressions

Not all Asset Expressions can be converted into Asset instances. There are several restrictions which follow from the conversion and typing rules presented above. An expression e is convertible under the following two conditions:

1. The expression must be well-typed in $\mathrm{AE}_{\rightarrow}$. This has two important consequences which are necessary for the conversion of expressions. The first affects types: Being member of a class is strictly necessary for Asset instances, as the class defines the structure of the instance. For this reason, it is important for the Asset Expression to provide semantic types which can—via the type mapping functions—be used as a basis to construct Asset classes. The second effect is of structural nature: Expressions which do not have a corresponding abstraction for each application cannot be converted by rule 6.2. They are also not well-typed in $\mathrm{AE}_{\rightarrow}$.

2. All explanations in e must be complete. In other words, there must be applications on all abstractions.

   Condition 1 and 2 together ensure that rule 6.2 can convert all explanations. The completeness of explanations is not covered by well-typedness even though availability of abstractions for each application is. The former is well-typed in $\mathrm{AE}_{\rightharpoonup}$ (it is of a function type), while the latter cannot be typed in $\mathrm{AE}_{\rightharpoonup}$ as the typing rule for applications (rule 3.1e on page 65) requires the operator to be of function type.

The fact that only a subset of all possible Asset Expressions can be converted raises the question of what is lost with those expressions that cannot be handled. Condition 1 rules out expressions that have applications not matched by any abstraction. Applied expressions explain some aspect of the explained content. Which aspect this is, is denoted by the corresponding abstraction (and possibly a content component). However, if no such abstraction exists, it cannot be determined whether the applied expression has anything to do with the content and if so, what their relationship is. Expressions of this group are therefore mis-constructed which is also reflected by their ill-typedness in $\mathrm{AE}_{\rightharpoonup}$.

The second requirement is that explanations must be complete. While Asset Expressions can represent incomplete information—for example as an intermediate representation in a workflow (see section 5.4.1) which can be completed in later steps—class-based systems generally require bindings for all attributes of a class. This is usually reflected by some kind of *new* operator or function that is responsible for creating new instances in one atomic step (see, e.g., [AC96, page 74]). Asset classes are no exception to this rule, therefore requiring convertible Asset Expressions to supply the necessary values.

Chapter 7

Pragmatics

In the previous chapters, Asset Expressions and their type systems have been introduced. Expressions can be generically managed in Asset Expression Systems, which focus on providing services for a medium number of users on their personal expressions. This chapter introduces services for type inference to aide large-scale handling of Asset Expressions. Both services use AESs as their basis.

The first service, which is described in section 7.1, does type inference in $AE_{\rightarrow}$. It uses semantic types of abstractions to infer a semantic type for the content that was abstracted from. It uses as input the typing decisions that have been made in previous expressions by comparing the sets of types from their abstractions. Its result is a ranked list of semantic types. The highest ranking type can either be used automatically or the creator of the new expression can pick a type from the ranked list.

The second service, which is described in section 7.2, infers structural types from AE_C. It creates Asset classes to type a set of Asset Expressions intensionally. The constructed classes form a conceptual schema and can be used as a transition path from AESs to schema-based software systems that allow uniform handling of mass data.

7.1 Type Inference for Semantic Types

The semantic type of content is assigned by the creator of the expression. In general, this semantic type is independent of the explanations, compare chapter 3.3. However, it is reasonable to assume that some expressions from an application domain, which share a common semantic type, are likely to contain similar explanations. In a scenario where users are building new expressions to add to an application domain that is already richly modeled by a large number of expressions, similarities in explanations can be exploited to suggest possible semantic types for new expressions.

More concretely, this support applies to situations where the user has created a number of abstractions over a piece of content but has not assigned a semantic type to the content yet. In this section, the possibilities of suggesting such a semantic type based on the signature of the overall expression are discussed. The signature is defined as it would be for computational functions: with a number of typed parameters—given in the abstractions—and a return type—the semantic type of the content. Applications on the abstractions of the expression are not taken into account as they are not part of the signature. The same is true for the names of the abstraction variables. The problem of finding appropriate semantic types is thus mapped onto the problem of matching function signatures based on the types of their formal parameters. The signature of the new expression is compared to the signatures of all existing expressions to find the most similar ones.

Traditional signature matching [MW97] takes into account exact matches of signatures as

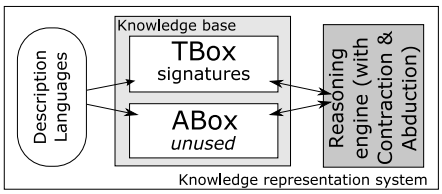


Figure 7.1: *Knowledge representation system. Figure adapted from [BMNPS04].*

well as relaxed matches where the signatures have a certain degree of similarity. If the types of the formal parameters from both signatures are pairwise identical, the signatures are considered an exact match. Possible relaxations are the generalization and specialization of types, the (un)currying of functions, and the reordering of tuples [MW97]. Depending on the application, one can include parameter names in comparisons or consider them of low importance and thus define equality of parameters as equality of their types.

The signature matching presented here encodes the signatures in a description logic and employs the non-standard inferences of contraction and abduction (see, e.g., [CCC+04]) to implement the relaxed matches. Matching is performed solely on the types of parameters. The following kinds of matches are considered:

- *Exact.* Two signatures which have the same list of formal parameters without regard to their ordering are an exact match.
- *Generalization and specialization.* If an existing signature has a formal parameter of more general or specific type this is a case of generalization or specialization, respectively.
- *Modified cardinalities.* A parameter might occur more often or less often in an existing signature than in the new one.
- *Expansion and truncation.* If an existing signature has an additional parameter, this is an expansion. The reverse case is called truncation. Expansion and truncation can be considered special cases of modified cardinalities, but with regard to the impact they have on the signature, it makes sense to separated the two groups.

Generalization and specialization as well as expansion are handled by abduction, modified cardinalities and truncation by contraction. Both inferences are discussed in section 7.1.2 below.

The presented approach works by pair-wise comparison of the new signature to the signatures already present in the system. Each comparison results in a *penalty* that reflects the degree of difference of the compared signatures. The approach has been implemented and tested with different functions to compute the penalties [Mue07].

Besides the suggestion of semantic types for expressions, signature matching can also be used in the retrieval of expressions. The idea is then that users supply hypothetical signatures describing aspects of the sought expressions. Signature matching is used to find the expressions that best satisfy the hypothetical signature.

7.1.1 Representation of Signatures

Description logics are heavily used in knowledge representation systems (KRS, [BMNPS04]). These systems provide the means to set up and manipulate a *knowledge base*. They also offer reasoning services on the knowledge base. The knowledge base is typically composed of a terminology and assertions which are held in the TBox and ABox, respectively. The TBox defines a vocabulary used in the application domain. The ABox contains individuals according

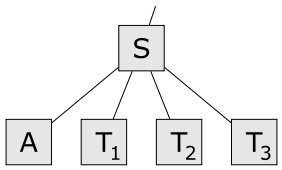


Figure 7.2: *Hierarchy of semantic types*

to this vocabulary. For the purposes of signature matching by contraction and abduction the ABox is not used. This general system layout is sketched in figure 7.1.

A key service provided by knowledge representation systems is determining the satisfiability of concepts provided in a TBox. A concept is called satisfiable if there exists a model in which the concept has individuals. Another important service of the reasoning engine is the computation of the subsumption hierarchy of all concepts in the TBox. Both satisfiability and subsumption hierarchies are used in contraction and abduction.

An $\mathcal{ALEN}$ [BMNPS04, chapter 2] logic is used to model signatures. As a basis for signatures, the hierarchy of semantic types is first introduced to the terminology box of the reasoner. This model for the hierarchy of semantic types is based on [BB04]. For each semantic type A a subsumption relationship to its supertype S is introduced:

$$S \sqsupseteq A$$

As it is assumed that no two semantic types are identical, appropriate disjointness rules are added for each type A:

$$A \sqsupseteq \neg T_i$$

where T_i are the siblings of A. These relationships are depicted in figure 7.2.

The general signature concept that is refined for all particular signatures is defined as:

$$\text{sig} \equiv \exists_{\geq 1}\text{hasParameter}.\mathit{Any} \sqcap \exists_{=1}\text{hasReturnType}.\mathit{Any} \tag{7.1}$$

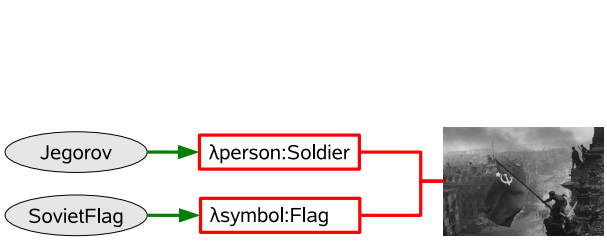
Concrete signatures are then modeled as specializations of this concept to reflect the actual parameters and return type of these signatures by adding appropriate hasParameter relationships. For each parameter a cardinality term is introduced. Thus, a signature with a single paramter T which is a direct subtype of Any will be modeled as:

$$\text{sig} \sqcap \exists_{=1}\text{hasParameter}.\mathit{Any} \sqcap \exists\text{hasParameter}.T \sqcap \exists_{=1}\text{hasParameter}.T$$

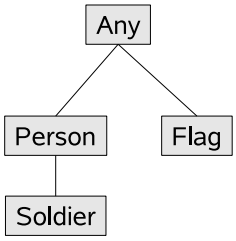
For each parameter, two conjunctive terms occur. The first to describe the type of the paramter. It is of the form $\exists$hasParameter.T for a type T. The set of these terms for a signature concept *sig* is *params*(sig). The second term describes the cardinality of parameters of this type. This term takes the form $\exists_{=n}$hasParameter.T for a type T with cardinality n. The set of these cardinality terms is *cards*(sig). The parameter terms are redundant but facilitate the specification of the contraction and abduction algorithms.

As parameters are identified solely by their type. Their *cardinality* is defined as follows: The cardinality c of a type is the sum $c := c_d + c_s$, where c_d is the number of parameters that are of this type directly, and c_s the number of parameters that are of (transitive) sub-types. The cardinality condition has to be met on each level of the semantic type hierarchy, even if no direct parameter of this type is present. Therefore, the signature concept will always include terms of a complete path of semantic types which extends from the top of the hierarchy to the specific types used in the signature.

The signature concept can be described as a compact graph. This graph is a subgraph of the hierarchy of semantic types that is annotated with the cardinality of each level as the sum



(a) Asset Expression



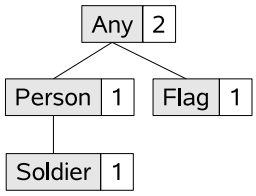
(b) Type hierarchy

$$\begin{aligned}\text{sig}_A \equiv \text{sig} \;\sqcap\; & \exists_{=2}\text{hasParameter}.\mathit{Any} \;\sqcap \\ & \exists_{=1}\text{hasParameter}.\mathit{Person} \;\sqcap \exists_{=1}\text{hasParameter}.\mathit{Soldier} \;\sqcap \\ & \exists_{=1}\text{hasParameter}.\mathit{Flag}\end{aligned}$$

(c) Signature concept

Figure 7.3: *Example of an Asset Expression signature modeled in description logic.*

of parameters at or below this type. For each parameter type, it shows all types on the path to the topmost type Any. Using the example from figure 7.3(a) this graph is:



In this example, no parameter is of a supertype of another parameter type. Therefore, the annotation of inner nodes is the sum of their children. Figure 7.3(c) expresses the corresponding signature in the DL-based model.

In addition to the direct model of a signature, a model for the *callability* of this signature from a context can be defined. To call a function with a signature $\prod_i(\exists_{=n_i}\text{hasParameter}.T_i \sqcap \exists\text{hasParameter}.T_i)$ the calling context has to have available at least n_i parameters of each type T_i. This can be expressed separately in a model for the callability which is very similar to the signature model. The difference is in the laxer restrictions on the cardinalities that call for a lower bound instead of an exact number:

$$\exists\text{hasParameter}.T_i \sqcap \exists_{\geq n_i}\text{hasParameter}.T_i$$

are the conjuncts for each parameter type T_i.

7.1.2 Reasoning for Signature Matching

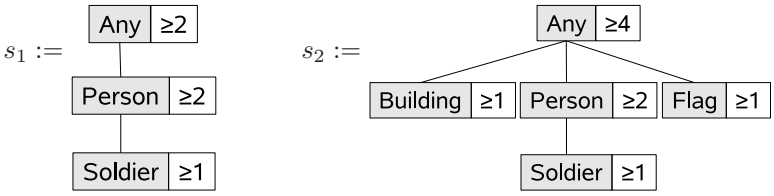
Using the signature and callability models presented above, callability concepts for all Asset Expressions can be defined in the TBox of the KRS. Given these concepts, the task is to match the signature of a new expression against each of the callability models. If the match is an exact one, this task is simple because it maps directly to the service for the computation of the subsumption hierarchy in the KRS. If inexact matches are also desired, more sophisticated inferrence services have to be employed. How this is done will be described in this section.

Below the signature of the newly created expression is called the *demand* signature C_d, the callability models which are already available in the system are called the *supply* signatures

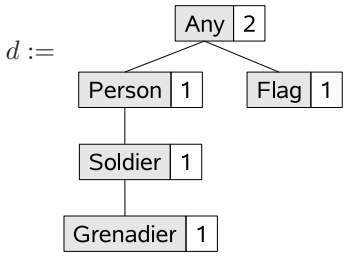
C_s. This is analogous to the terminology of [CCC+04, CNS+04]. In the presented scenario the supplies are fixed because they correspond to defined Asset Expressions which cannot be changed as a result of the matching process. Therefore, contraction and abduction modify the demand C_d. Contraction relaxes the cardinalities of parameters (potentially up to the point of removing the type alltogether from the signature), while abduction adds hypothetical conjuncts to the supply. In literature the concept contraction problem is sometimes defined as modifying the supply. However, the problem is symmetric and can be defined exchanging supply and demand [CNS+04].

Using concept contraction and concept abduction, signatures can be found which are similar to the demand signature C_d. The degree of difference is described by the penalty π which increases for each modification that has to be made to the demand in order to match the supply. The result of a match is a list of all callability models (and attached to them Asset Expressions) that is ordered by the respective penalty.

The examples in this section use the supply signatures s_1 and s_2 which are described by the following callability models:



Signature s_1 has two parameters, one of type Person, one of type Soldier. The second signature s_2 has four parameters one Person, one Soldier, one Building, and one Flag. The demand d is a refined version of the signature shown in figure 3.7:



Contraction

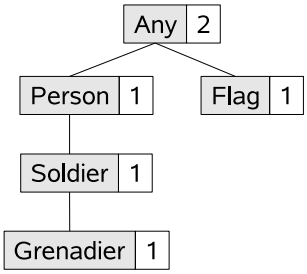
A signature from the supplies might contain parts that are contradictory to the demand signature. These parts are removed by concept contraction of the demand in order to discover which part of the demand can be kept as it is compatible with the supply. More specifically, a concept contraction problem [CNS+04] (CCP) is to split the demand C_d into a pair of concepts K and G such that the conjunction of K and C_s is satisfiable:

$$\text{CCP}: \quad sat(K \sqcap C_s), \qquad C_d = K \sqcap G$$

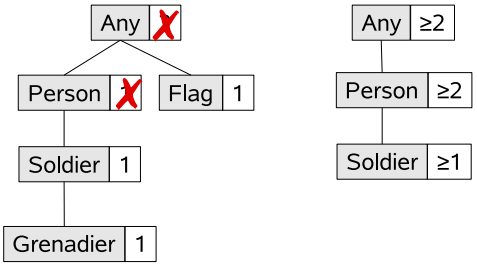
K and G are called the *keep* and *give-up*, respectively. The goal is to find conjuncts K that are compatible with the demand. A penalty is calculated depending on the give-up G.

Concept contraction has been applied to several other areas, for example the matching of user profiles [CCC+04]. While the algorithm is essentially the same in all application domains, it has to be adapted to the particular model of concepts to be contracted. Especially important is the design of a custom penalty function.

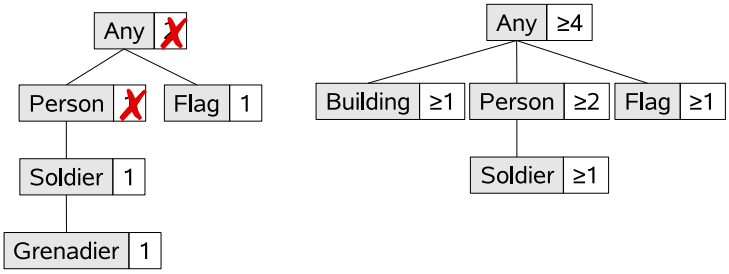
The purpose of contraction when applied to models of signatures is to remove contradictions by adjusting the cardinalities appropriately. Other reasons for unsatisfiability of $K \sqcap C_s$ cannot



(a) Demand (figure 7.3)



(b) Keep and Supply s_1



(c) Keep and Supply s_2

Figure 7.4: *Two examples of concept contractions. In the first example an inner cardinality does not match the supply. In the second example the total number of cardinalities is wrong, but the cardinalities of the leaves remain in the keep.*

occur as there is no way to define "un-parameters" in signatures which would explicitly forbid a parameter of a certain semantic type to be present.

The concept contraction algorithm works on the conjuncts that define the cardinalities in the signatures. For each pair of conjuncts from demand and supply that describe the same type, the compatibility of the cardinalities is checked. If the terms are contradictory, the term from the demand is removed and the penality increased accordingly. The contraction penalty π_C takes into account the relative difference between the cardinalities in the demand n_d and supply n_s for the given type. It is therefore defined as a function of these cardinalitites:

$$\pi_C(n_s, n_d) := \frac{n_s - n_d}{n_d} \tag{7.2}$$

Intuitively the idea is to define in relative terms how far the demand is from the supply, i.e., how large a change would be necessary on the demand to meet the supply.

Using this penalty, the concept contraction algorithm is given in pseudo code below. It works on a demand C_d and a supply C_s by comparing cardinalities on types that occur in both:

foreach $c_d := \exists_{=n_d}\text{hasParameter}.T_d \in cards(C_d)$ do
  foreach $\exists_{\geq n_s}\text{hasParameter}.T_s \in cards(C_s)$ do
    if $T_d = T_s$ and $n_d < n_s$
      remove c_d from C_d
      $\pi := \pi + \pi_C(n_s, n_d)$
      removeSuperCards(K)
    end
  end
end

The cardinality of a node always has to be at least the sum of its children. The function *removeSuperCards*(K) therefore removes the cardinality terms for all super types to prevent unsatisfiable models. Two examples of concept contraction are shown in figure 7.4. In the first example (figure 7.4(c)) the top-level cardinalities as well as the cardinality of Person in the demand contradict the supply and are therefore removed. In the second example the top-level cardinality happens to match and is therefore kept, even though the number is obtained differently in supply and demand. Since there is one Grenadier there is also one Person. However, the supply calls for at least two, the term is thus removed from the demand. During contraction only the cardinality terms are removed.

Abduction

Concept contraction has split off a keep K from the demand signature, such that the conjunction of K and the supply is satisfiable. There might, however, be parameters in the supply that are not present in the demand. It is the aim of concept abduction to find a hypothesis that—when added to the demand conjunctively—makes the supply subsume the demand. In the model of signatures used here, the subsumption of the demand by the supply translates into the possibility of making a "function call" with respect to the signature of the supply by using the parameters from the demand. Some of the parameters that are necessary to do this might be missing, they are added during abduction.

More formally, the concept abduction problem [CNS+04] (CAP) of a supply C_s and a keep K is:

$$\text{CAP}: \quad C_s \sqsupseteq K \sqcap H$$

Where H is the hypothesis to be added to the keep. Intuitively, the supply requires a parameter that the demand does not yet provide. This parameter is hypothetically added to the demand as further investigation might reveal that it indeed exists in the context of the demand. In an Asset Expression this means that an additional abstraction over the content is sensible but has not been made.

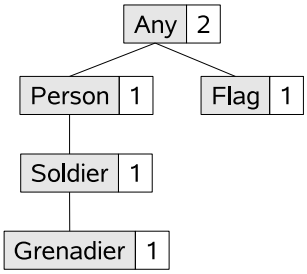
Similarly to concept contraction, a penalty is also applied in this case for each hypothetical parameter. The penality value π_A for an additional term that has the cardinality n_s on a supply that has n_{Any} parameters is calculated as (n_d is the cardinality on the demand as before):

$$\pi_A(n_s, n_d, n_{Any}) := \frac{n_s - n_d}{n_{Any}} \tag{7.3}$$

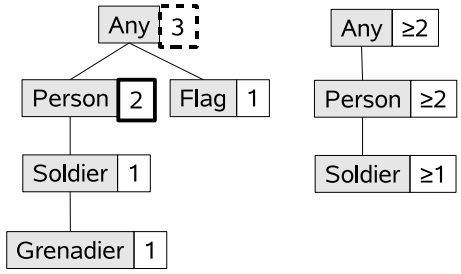
If the term did not exist in the demand n_d is 0.

The hypothesis H can then be calculated for a supply $C_s = \ldots \exists_{=n_{Any}}\text{hasParameter}.Any \ldots$ and a keep K as follows:

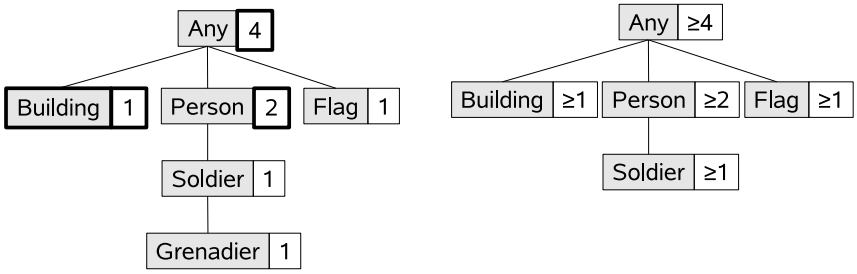
foreach $p_s := \exists\text{hasParameter}.T_s \in params(C_s)$ do
  if there exists $p_d := \exists\text{hasParameter}.T_s \in params(C_d)$
  then
    if not $c_s := \exists_{\geq n_s}\text{hasParameter}.T_s \in cards(C_s)$
    then



(a) Demand (figure 7.3)



(b) Keep with Hypothesis and Supply s_1



(c) Keep with Hypothesis and Supply s_2

Figure 7.5: *Two examples of concept abduction. The hypotheses are added to the keeps with bold frames. In the first example, a computed cardinality has to be adjusted as a result of a hypothesis.*

add c_s to H
$\pi = \pi + \pi_A(n_s, n_d, n_{Any})$
fixCards(K)
end
else
add p_s to H
add $c_s := \exists_{\geq n_s}\text{hasParameter}.T_s \in cards(C_s)$ to H
$\pi = \pi + 2\pi_A(n_s, n_d, n_{Any})$
fixCards(K)
end
end

Some of the cardinalities in the signature model of the keep are computed as the sum of their children. These cardinalities reflect the overall structure of the signature and need to be kept up-to-date during abduction. This bookkeeping is done in *fixCards*(K) which adjusts the

cardinalities bottom-up to account for potential violations introduced in the last abduction step. As witnessed by the example in figure 7.5(b) these violations do not necessarily happen.

All parameter terms in the supply are checked for existence in the keep. For parameter terms which are also found in the keep, the next step is to ensure the existence of the corresponding cardinality term. It is guaranteed by concept contraction that if a cardinality term exists in the keep, its cardinality is compatible with the supply. If no cardinality term is found in the keep, the term from the supply is added and the penalty increased by π_A accordingly.

For parameters which are not present in the keep at all, both the parameter and the cardinality term from the supply are added. The penalty applied in this case is twice the penalty π_A.

Figure 7.5 shows the results of the abduction phase for the two examples from figure 7.4. The hypotheses are added to the keeps in bold. The first example has become structurally very similar to the supply, just one parameter is modeled more specifically in the demand. The second example required an adjustment of computed cardinalities which is shown with dashed frame. This is the case as the Flag parameter could be retained (it does not contradict anything from the supply).

7.1.3 Example

This section presents an example of semantic type inference by contraction and abduction. Due to space constraints, the set of supplies is very small: just four expressions. The example is based on the hierarchy of semantic types shown in figure 7.6. The figure should be considered to show an excerpt from a larger, realistic type system. Abbreviated expressions for the supplies are shown in figure 7.7 in the left column. The expressions omit all applications, which are likely to exist on real-world supplies, as these are not taken into account by contraction or abduction. The right column of figure 7.7 shows the callability models of the supplies as introduced above. The supplies are matched against a new expression shown in figure 7.7(e) along with its signature model. The content in expression 7.7(e) has not been assigned a semantic type yet. It is the purpose of semantic type inference to suggest appropriate candidates.

Figure 7.8 shows the results of contraction and abduction on the demand for each of the supplies. The graphs in the left column are a combination of those used previously to illustrate contraction and abduction separately. As before, the demand is modified as the supplies already exist and cannot be changed to match a demand. In the demand, cardinalities removed by contraction are crossed out. Terms added to the hypothesis by abduction are shown with bold frame. Cardinalities which had to be adjusted to keep the model consistent are crossed out with a single line. The new cardinality is shown in a dashed box. The corresponding penalties are also shown.

The penality π_a is computed as follows:

$$
\begin{aligned}
\pi_a = 1\frac{1}{30} = \frac{6-5}{5} &\qquad \text{removal of cardinality on Any} \\
+ \frac{6-5}{6} &\qquad \text{new cardinality for Any} \\
+ 2 \cdot \frac{1-0}{6} &\qquad \text{introduction of Vehicle} \\
+ 2 \cdot \frac{1-0}{6} &\qquad \text{introduction of MilitaryVehicle}
\end{aligned}
$$

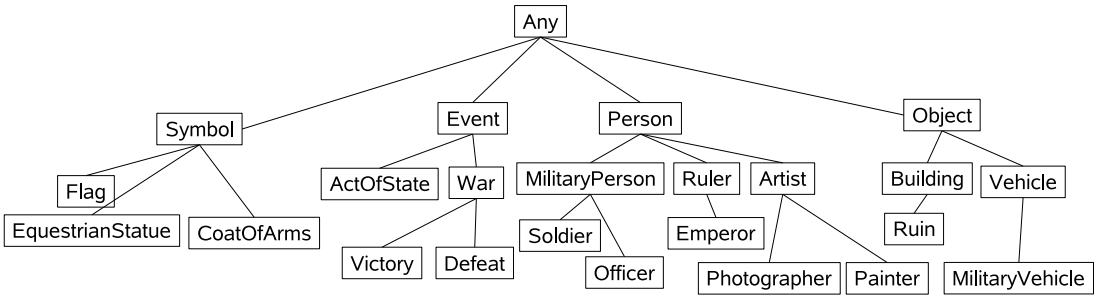


Figure 7.6: *Semantic type hierarchy for inference example. Supertypes are shown above their subtypes.*

The penality π_b is computed as follows:

$$\begin{aligned}
\pi_b = 1\frac{1}{2} = \frac{2-1}{1} &\qquad \text{removal of cardinality on Object}\\
+\frac{2-1}{5} &\qquad \text{new cardinality for Object}\\
+2\cdot\frac{1-0}{5} &\qquad \text{introduction of Painter}\\
+2\cdot\frac{1-0}{5} &\qquad \text{introduction of Animal}
\end{aligned}$$

The adjustment for cardinalities for purposes of model consistency (on Any, Person, and Artist; dashed boxes) does not incur a penalty.

The penality π_c is computed as follows:

$$\begin{aligned}
\pi_c = 1\frac{2}{3} = 0 &\qquad \text{nothing changed in contraction}\\
+2\cdot\frac{2-0}{3} &\qquad \text{introduction of Ruler}\\
+2\cdot\frac{1-0}{3} &\qquad \text{introduction of Painter}\\
+2\cdot\frac{1-0}{3} &\qquad \text{introduction of Emperor}
\end{aligned}$$

The penality π_d is computed as follows:

$$\begin{aligned}
\pi_d = 1\frac{4}{5} = \frac{1}{2} &\qquad \text{removal of cardinality on Object}\\
+\frac{1}{2} &\qquad \text{removal of cardinality on Building}\\
+\frac{2-1}{5} &\qquad \text{new cardinality on Object}\\
+\frac{2-1}{5} &\qquad \text{new cardinality on Building}\\
+2\cdot\frac{1-0}{5} &\qquad \text{introduction of Painter}
\end{aligned}$$

As π_a is the lowest penalty, the expression in figure 7.7(a) has the signature that is most similar to the demand. The semantic type Victory is therefore the best candidate for a typing of the content in figure 7.7(e).

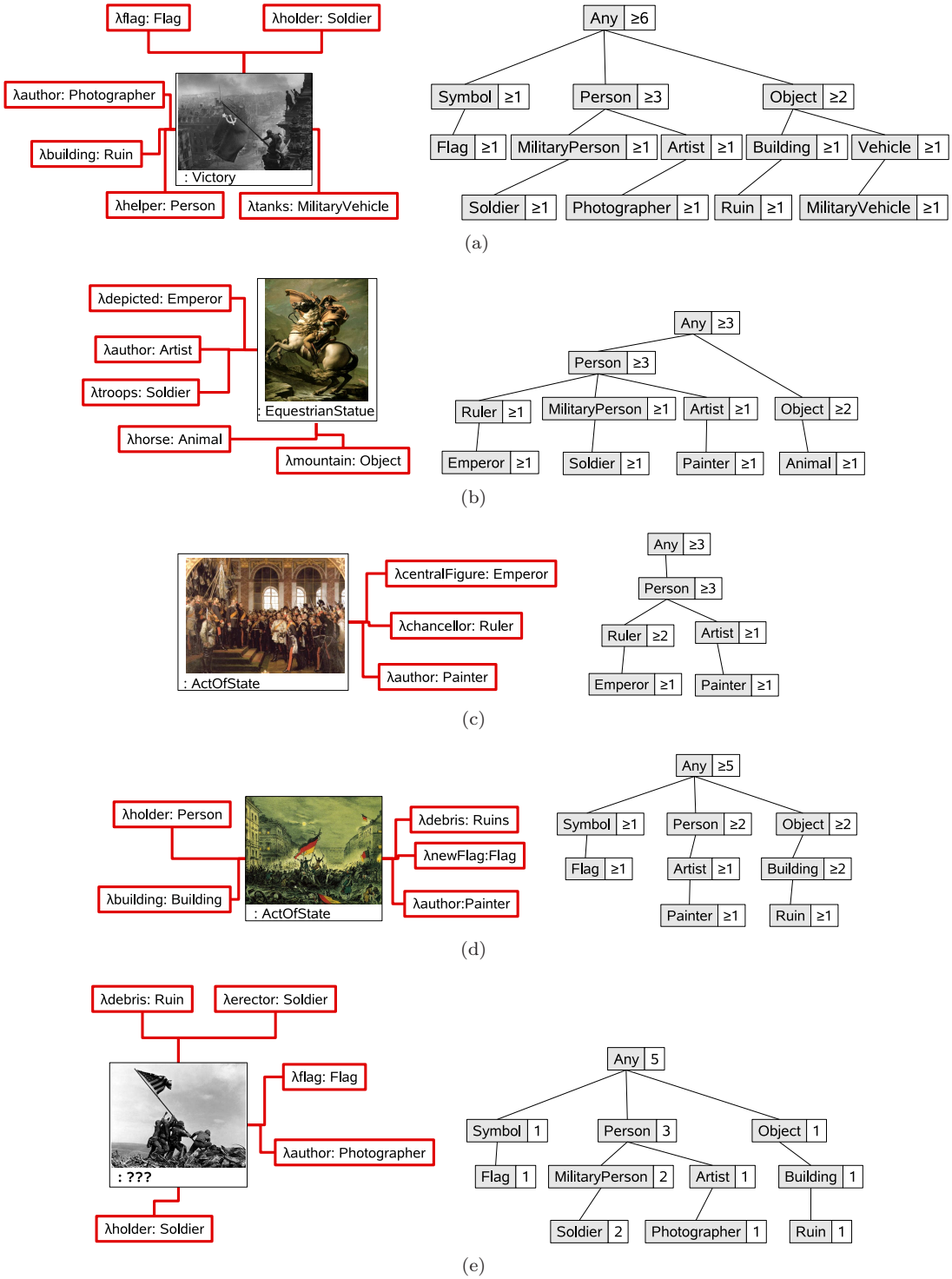


Figure 7.7: *Four Asset Expressions (a) through (d) as a basis for the semantic the type inference example. One new expression (e) for which a semantic type is to be inferred. Expressions are shown along with their signature or callability model expressed as an annotated sub-tree of the semantic type hierarchy. The annotated number is the total number of parameters of the annotated type and its children.*

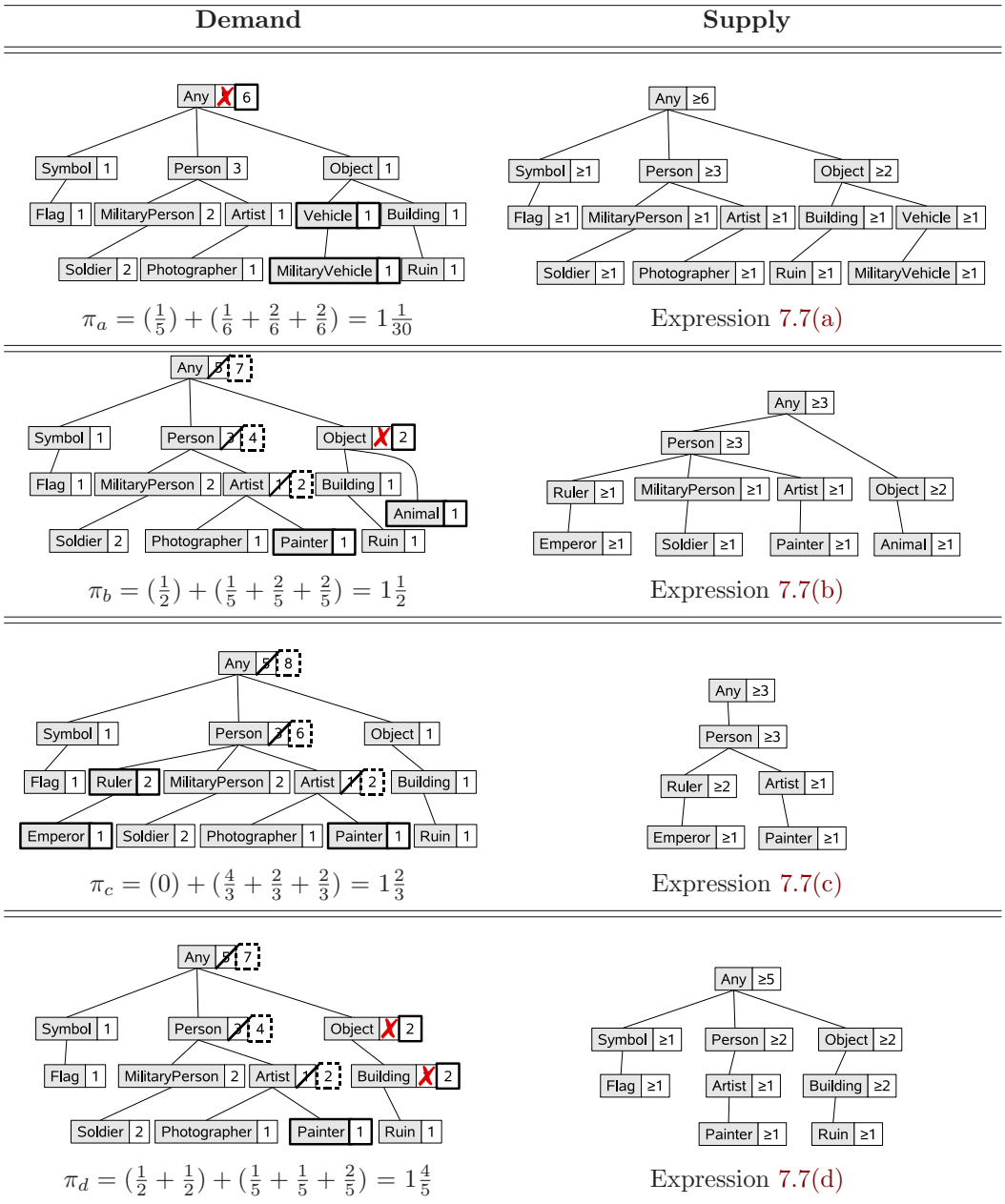


Figure 7.8: *Results of contraction and abduction in the semantic type inference example. Modifications to the demand are shown in the type tree.* ✗ *marks cardinalities removed by contraction, bold boxes* ☐ *mark hypothetical terms from abduction, adjusted cardinalities are indicated by dashed boxes* ⬚. *The penalties π are shown as the sum of contraction and abduction penalties $(\pi_C) + (\pi_A)$.*

7.2 Type Inference for Intensional Types

The class-based type system AE_C presented in chapter 6 lays the foundations to create a conceptual schema of an application domain that is modeled in Asset Expressions. When new expressions are created that call for a different schema—for example due to the inclusion of the personal opinion of a user or a better understanding of the application domain—a new schema can be created via AE_C to reflect these changes. Two challenges remain:

1. Summarizing the numerous individual classes which are used to type each expression into a coherent conceptual schema of the application domain. If the underlying Asset Expressions are totally uniform, this task is trivial, but one can generally not expect such conditions.

2. Controlling the evolution of the conceptual schema in successive iterations by helping users to understand the impact of their action on the schema. This is important because the conceptual schema is the basis for an information system which is populated with data. This data must certainly remain accessible across evolution steps, but many changes to the schema are conceivable that limit such access.

To meet these two challanges a process to support schema creation from Asset Expressions will be described in this section. It is called the *User Centric Schema Creation Process* (UCSCP). While the process allows users to make changes to an open schema and have the information system react dynamically to those changes in the spirit of Conceptual Content Management (section 2.1.4), it helps users by guiding their modeling decisions. It does so by ensuring that schema changes are well-founded (i.e., based on a substantial number of instances) and compatible with the existing schema. Practical experience—a brief account is given in the next section—has shown that these are central issues when domain experts model an application domain.

7.2.1 Lessons Learnt

Conceptual content management systems are considered open if they allow their users to work with a schema that matches the particular user's needs. Therefore, users must be able to modify the schema according to their requirements at any time. Such schema modification abilities are most useful if they have an immediate effect on the system. In particular it is felt that a manual development round-trip—consisting of writing a change request, a brief analysis, design and the subsequent implementation—is not appropriate as it causes a too high latency for the user experience to be truely open. Instead the system itself must be dynamic, meaning that it can adapt to the schema modifications of users on its own and almost in real-time. It is dynamics that put users in the position to flexibly model the application domain according to their needs and opinions while remaining in a schema-based system. Users can make modifications successively, improving the schema as they move along. This is a large benefit for many user groups who are not at ease with abstracting from an application domain to create a complete domain schema before they can fill the schema with instances.

In an application project a high number of reproductions of documents from archives were to be collected to enable future research on the domain of art history. Some more details on the project and a brief introduction to the application domain can be found in chapter 8. Due to the large amount of data which was to be expected, an information system was set up to support the collection process. One of the first tasks of the project was therefore to create a conceptual schema to describe the documents. An initial survey of available documents made clear that these documents were quite heterogenous. Therefore, a number of document classes were introduced. The creation of these classes proved difficult as on the one hand the involved computer scientists did not fully understand the nature of the documents and were therefore not able to conceptually capture all important parts. On the other hand the creation

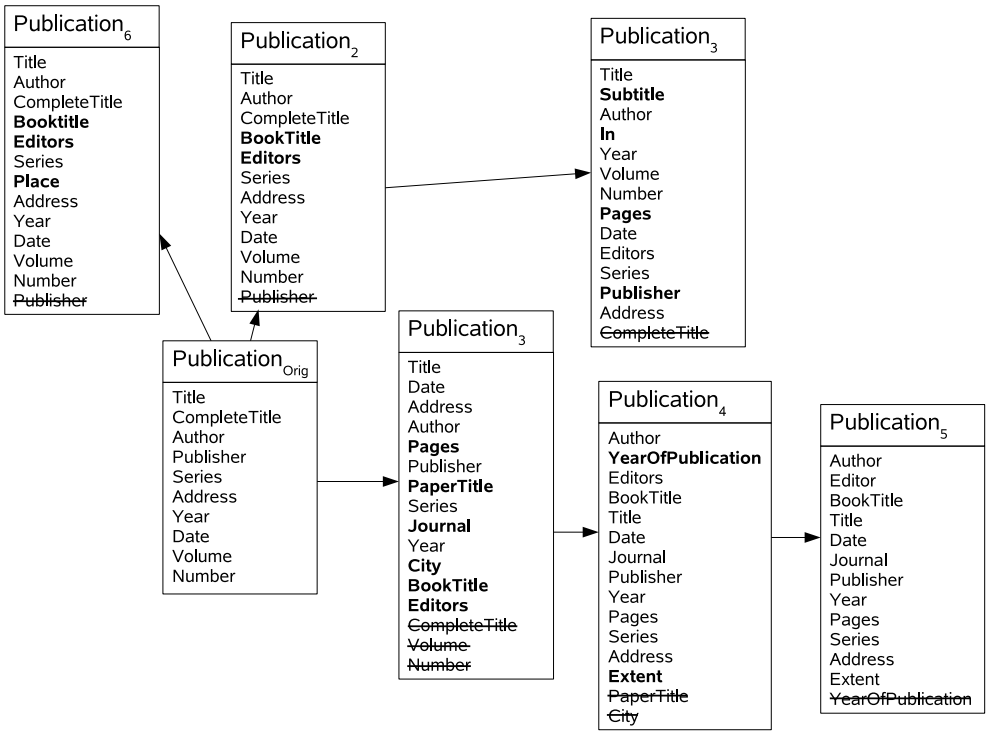


Figure 7.9: *A class from the GKNS project and the various forms it would have been in, had the users had full openness and dynamics. The arrows denote different branches of evolution. New attributes are shown in bold, removed attributes are crossed out.*

of conceptual schemata is not amongst the usual tasks for art historians who thus had trouble abstracting from a large set of documents to create a joint schema.

After some discussion an initial schema was agreed upon and a prototype of a data entry tool developed that worked according to this schema. The idea was then to let the domain experts input some documents to evaluate how well these could be mapped with the given schema. The schema was then expected to be modified according to the advice given by the domain experts. The schema was therefore developed in an open and dynamic way. Openness was not on an individual basis, as the result had to be a general schema for the whole of the application and dynamics was deliberately not immediate to force the group to reflect on changes proposed by members.

This experiment exposed two interesting points:

1. As the propositions for changes were made by individuals, they always reflected a personal opinion. Had the process been open and dynamic for each of the users, these changes would have immentiately caused a new and personalized variant of the system to be created. However, it was obvious that already after one month these variants would have been so different schematically that re-integration as well as exchange of data would have been impossible (compare the personalizations shown in figure 7.9). This prompts for some control over openness and dynamics which alerts users to such problems. After all, most information systems—and conceptual content management systems put particular emphasis on this—need to allow users to collaborate. This is the primary reason why much effort is spent to implement personalization in a way that retains access to existing data instances [RR98, SBS06].

2. Changes were usually requested on the basis of single documents. A common case for this was the appearance of a document which required a conceptual model slightly different to what was available to be described appropriately. Such a new document was then met with the proposition of a conceptual model ideal for it and this new model was proposed as a change to be integrated into the new schema. Little regard was given to the appropriateness of this new model in general or in the context of other existing documents. Had the process been truly open and dynamic, a high degree of variation would have been observed (figure 7.9) between such personalized schemata and their predecessors. It can be observed that personalization was often driven by a small number of—usually a single—sample instances. To avoid problems with representing existing instances, personalization should be put into context of a larger amount of instances.

The unrestricted availability of openness and dynamics places a large responsibility on the user. Their proper use thus requires a high degree of skill in conceptual schema creation which should not be assumed to be present with domain experts [SM99]. The UCSCP therefore aims at sufficiently guiding its users to allow domain experts to usefully personalize conceptual schemata without having to understand their theoretic foundations or having a background in domain modeling. Additionally, it should be stressed that the conceptual schema was not understood by the domain experts when presented in an abstract fashion (e.g., in UML class diagrams). This is an observation also made by others [SM99, GWG06]. Transparency of the conceptual schema is, however, vital in order to receive useful feedback from domain experts [MK98].

7.2.2 Process Support for Openness and Dynamics

As discussed above, the following observations can be made about the involvement of domain experts in conceptual modeling:

1. Domain experts find it helpful to base their modeling descisions on examples. Some do this to such a high degree that it limits abstraction.

2. When deriving modeling decisions from examples, often too few examples are considered.

3. Conceptual modeling by domain experts carries a high risk of modeling errors in general [AW85] due to unfamiliarity with the concepts of modeling.

A proposed remedy [AW85] to these problems is the involvement of modeling experts. Unfortunately, this is not a feasible solution in open and dynamic systems. While the involvement of a modeling expert at least partially maintains the aspect of openness, that of dynamics is much dimished as each evolution step requires the interaction with the modeling expert who is usually not available at all times and on short notice.

To alleviate the situation Asset Expressions can be used as a flexible means to entitiy modeling. Only few modeling concepts have to be understood to create useful expressions in an application domain. These expressions can then be used by the AES to help its user to arrive at a conceptual schema of the domain described by the expressions. Together with a facility to import instances from an existing CCMS, the process can be used to support open model changes and dynamic system evolution in the CCMS. Without CCMS, the process can be used to create a conceptual schema, which then could be used to bootstrap any information system, for example a CCMS.

Contemporary methodologies for schema creation in software engineering assume the presence of a modeling expert to run the process. This expert works with someone knowledgeable about the application domain. In software development, the interaction between the two commonly is found in the requirements analysis phase during software development [DMFP05, MJF03, RMRD05, FKM04]. To support requirements analysis “structural approaches and own modeling languages have been proposed, but do not enjoy wide-spread adoption in practice”

([RP06, D13.5], author's translation). A major hinderance for the adoption of such languages is the difficulty of obtaining sufficiently formal statements of requirements.

There are models [TMP98] for interaction which make explicit how the modeling expert can obtain information relevant to modeling from the domain expert. The particular approach in [TMP98] is based on natural language descriptions from the domain expert and the modeling expert's ability to build formal models from these. A conceptual predesign phase [MK98] can be used to bridge between natural language texts and a conceptual model. Carrying out this mapping directly is a very complex task. This task is simplified by a dedicated predesign phase during which a minimal model is constructed. Parts of the predesign model can be constructed automatically from scenarios described in texts [FKM04].

Several approaches exist in literature that are related to the problem of extracting conceptual information from samples. The problem appears in many variants which have in common that information, which was created in an unstructured manner, needs to be captured in an implementation-level schema or conceptual model.

A text-based approach for semi-automatic ontology extraction from web pages [SCC05] relies on the pre-categorization of the documents. Documents are analyzed to extract relevant domain concepts. By means of the documents, these concepts are then clustered into a structured representation of the domain.

Similar to the UCSCP in its iterativeness is an incremental schema construction process [Pro97]. Its key idea is to treat schema construction as a sequence of model evolutions starting from a rough conceptual model and ending with an implementational database schema. The process requires modeling experise from the user as it is largely up to the user to arrive at the proper conceptual model to start the process.

While the UCSCP copes without a modeling expert to enable openness and dynamics, others [SGU02] aim to reduce the involvement of the domain expert. The idea is to encode general knowledge in a knowledge base which assists the modeling expert in building the schema. The modeling expert is asked to classify each term that appears during modeling in the ontology of general knowledge thereby giving the system hints about the application domain. An application of this approach to business system modeling [SCD+97] builds a knowledge-base about the application domain which organizes common facts. While the user builds the schema, a business reasoner makes recommendations on how to improve this schema. Structural comparison is used to identify equal or related descriptions.

The UCSCP to create a schema from Asset Expressions is shown as a UML activity diagram in figure 7.10. It consists of four phases which are shown with different backgrounds in the diagram. During phases I. and III. user interaction is required to create or import expressions as well as to give feedback on the generated schema. Phases II. and IV. are purely automatic construction of the schema and generation of the matching system, respectively. In the following the term "user" always refers to the domain expert.

The process contains three iterative elements. In the inner-most loop the user directly adjusts the presented schema (loop between phases III. and II.). This is a very direct way of giving feedback. A little less direct is the loop between phases III. and I. that allows the user to influence the schema by modifying the expressions that the schema creation is based upon. Finally, there is a loop from phase IV. all the way to the beginning of the process. This loop represents an open schema modification in a CCMS.

The UCSCP focuses on determining the structure of the application domain and documenting it in a conceptual schema. Behavioral aspects of a system supporting work in the application domain are also of interest. Means to involve the end user in their identification are available in the form of a process which extracts behavioral information from textual descriptions [FKM04]. It is based on a linguistic analysis of the description and carried out in the predesign model outlined above. Besides the goal of determining system behavior a key difference to UCSCP is that the approach relies on purely textual descriptions.

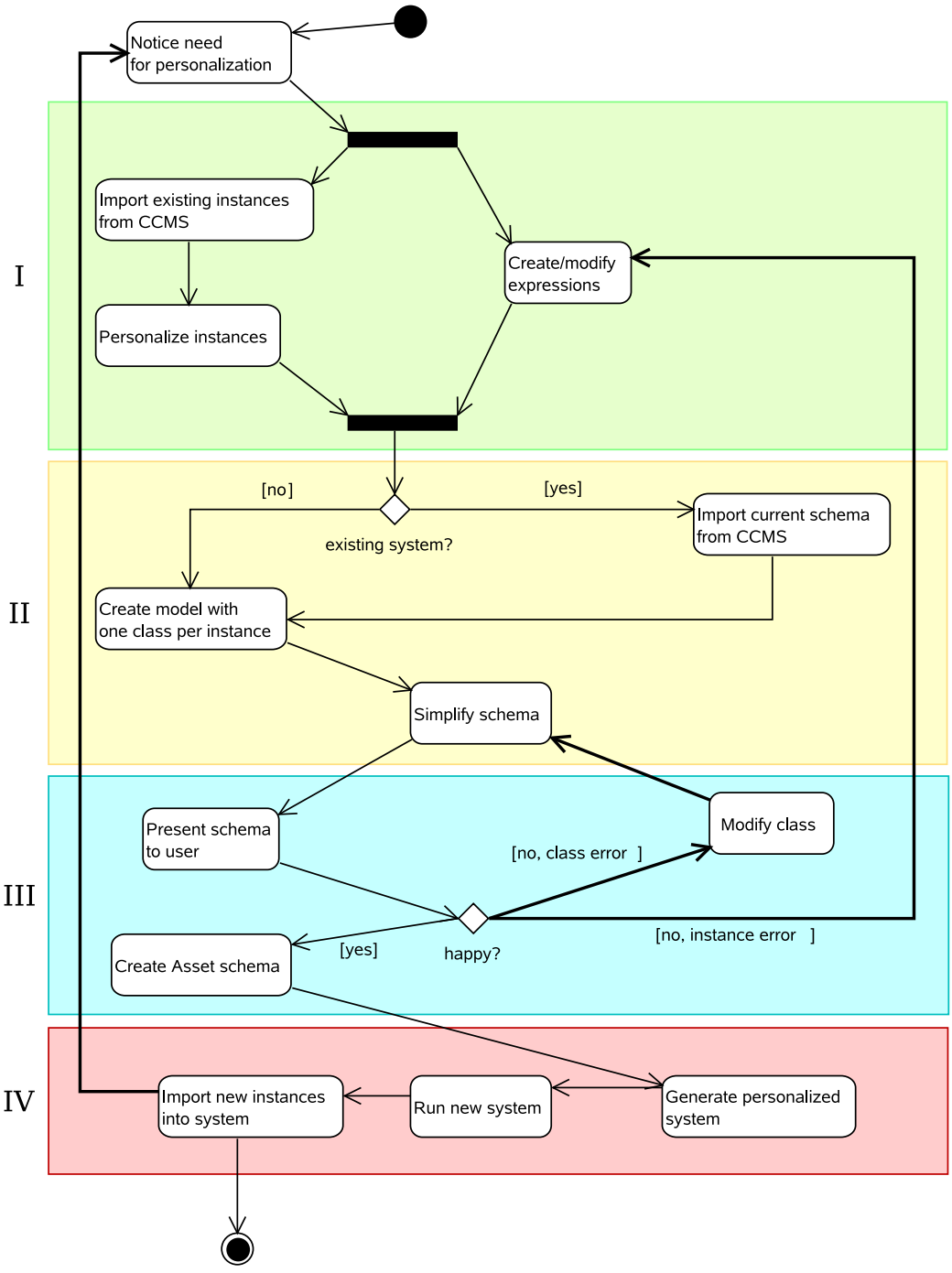


Figure 7.10: *An activity diagram of the User Centric Schema Creation Process to guide users in their use of openness and dynamics. It works in four phases: (I.) aquire instances, (II.) automatically create schema, (III.) collect and incorporate user feedback, and (IV.) generate system from schema. Feedback loops are shown as bold arrows.*

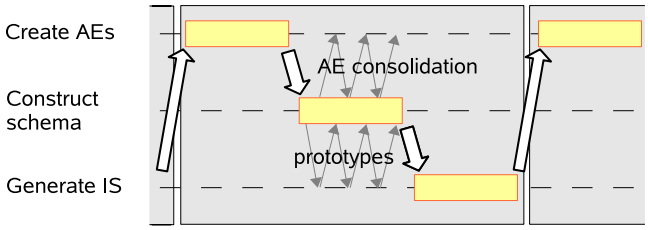


Figure 7.11: *Generation of an information system from Asset Expressions by construction of a conceptual schema which models the Asset Expressions.*

Phase I: Obtaining Initial Instances

The first phase is about obtaining Asset Expressions which are used as the basis for schema creation. There are two possible sources for these expressions: new creations by the user or imports from an existing CCMS. In the first case, the user has previously used the facilities of the AES to model entities with expressions and now selects from these expressions those that the schema is to be created from. In the second case, a number of instances are selected in a CCMS and the conversion from Asset instances to Asset Expressions as defined in section 6.2 is used. Either way, the result should be a comprehensive set of expressions from the application domain. Choosing appropriate expressions which best reflect the application domain is one of the central accomplishments of the user in this process. Expressions should be chosen such that the application domain is covered to a high degree with as few expressions as possible. However, there is a trade-off between the number of expressions and the ease of model creation: If only few expressions are used and among those expressions the ones that should be modeled by a single class are structurally diverse, schema creation becomes difficult.

To even out unnecessary structural diversity of related expressions, it is recommended that the user reviews the expressions after import or creation. This review does not have to take place in the first iteration but can be deferred until later to remove unwanted effects of structural diversity on the created schema.

If the process is used in support of open modeling in a CCMS, the user has the opportunity to make personal changes to the imported expressions (which still reflect the original schema) with the means of the AES before schema construction.

Phase II: Schema Construction

The second phase is that of automatic schema construction. The system infers a schema based on the given expressions. This is done in two steps: All expressions are typed in AE_C resulting in one Asset class per instance. As this is usually not the desired schema, the individual Asset classes are then analyzed structurally to find equivalent classes and to place them in a class hierarchy. The second step also considers Asset classes imported from an existing schema in the case of open model evolution.

The core of the second phase are therefore schema matching and inferences problems. As these are common problems, there is a large body of literature [AS83] on the subject. Thus the discussion here will be brief and will focus on aspects particular to this process.

Structural inference from instances has been considered in a variety of environments. Examples include object-oriented classes [PS91], semi-structured data in general [GW97, WMBS04], and XML documents in particular [GGR+00, NJ02, WS03, BSG+04]. As the more recent literature on XML documents is usually based on XML Schema which uses structural types for elements, its findings are also relevant here.

In general, several operations on the model have been identified [Ber03]. These include: *Match* which computes a mapping between two models, *diff* which determines the differences

between two models, and *merge* which merges two models by collapsing elements that are equal in both to one element in the new model.

Schema inference algorithms usually work by first building a data-structure that accomodates all sample instances presented to the algorithm. In a second step this data-structure is then simplified to remove redundancies. Many approaches also apply generalization heuristics. In the UCSCP, the data-structure is the set of all Asset classes which are created during the initial typing of instances in AE_C. Simplifications and generalizations for these classes are described below. The most important means to simplify the schema is the aggregation of equivalent classes (a *merge* operation). The process takes a conservative approach to aggregation and performs it automatically only for classes that are identical. Two classes from the schema can be in several relationships which prompt different reactions from the system:

- *Identical classes*: Two classes are considered identical if all their attribute names as well as the attribute types are equal. In this case the process reacts automatically and aggregates all instances from both classes under a common class. This constitutes a *match* and *merge* operation.

- *Inheritance*: A class introduces additional attributes but is otherwise identical to another class. The process automatically puts the two classes in an inheritance relationship. In this case it is important to use the most specific super class, i.e., the class with the least differences, to obtain a well-structured class hierarchy. The *diff* operation plays a key role in determining inheritances.

- *Type match*: Two classes have attributes of exactly the same types but the attributes have different names. While some inference or matching algorithms (usually fully automatic ones) consider this a match, in this process the two will only be aggregated with explicit permission of the user. Especially in sparse models (where classes have few attributes) coincidental type matches are very common (see, e.g., figure 7.12). Type matches are related to signature matches presented in section 7.1 in that they compare a set of types. However, a type match only occurs if the sets of types are identical whereas semantic type inference uses inexact signature matches.

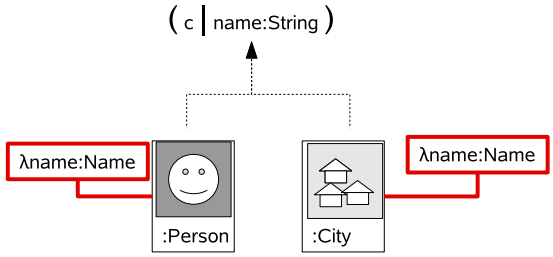
  It is not necessary to ask the domain expert directly whether two classes should be merged. Instead, two sets of Asset Expressions are shown and the question to the domain expert is whether these should be treated together or separately by the system. The effect of this decision can be observed in the prototype system constructed in the next phase.

- *Inheritance Orphan*: This is a special case of inheritance in which the subclass has only one (or very few, depending on the overall number) instance. Such a class is only kept in the schema with explicit permission of the user who is asked to supply further extensional backing for the class. If no additional instances can be found—either among the instances of the superclasses or as new expressions—it is recommended to remove the class.

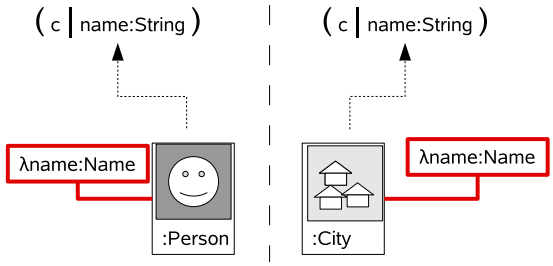
  Similar to the previous case, the question to the user is based on two sets of Asset Expressions. The first set contains the expressions from the super-class, the second the few instances from the orphan class. It is then up to the user to decide whether the difference in the few expressions warrants their special treatment. If the user determines this not to be the case, the system removes the additional abstractions. This has two purposes: to make the expressions typable in the super-class and to remind the user of the consequences of the decision, i.e., to make explicit the loss in information.

The first three cases have also been identified for matching of inensionally defined ontologies [HG02] as used for the creation of federated schemata.

Each of the classes needs a name to present to the user in the next phase. If all expressions of a class have a most specific common semantic type that is not Any, the name of this semantic type is used as a name for the class. It may be necessary to disambiguate class names, e.g.,



(a) Common class for structurally equivalent expressions.



(b) Class split in two.

Figure 7.12: *Expressions with coinciding structure and their allocation to classes.*

by adding appropriate suffices such as numbers. Classes for which the most specific common semantic type of all expressions is Any use "Class" plus a unique suffix as their name. If the process is used to support an incremental model evolution step, the number of new classes and attributes is typically low compared to the size of the original schema. It is then much easier to automatically create an appropriate schema requiring little user feedback in the next phase.

When automatically creating schemata either by inference from instances or other means, one is typically confronted with the problem of measuring the quality of the solution. Some approaches to the "goodness of fit" problem are discussed in [AS83]: The simplicity of the schema, the tightness of its fit on the sample instances, as well as several approaches to mix these measures. An important limitation of all measures is that they can only compare the schema against the available sample instances. However, while those instances might have been chosen to be as representative as possible, the intention of the user is usually not to receive a schema that describes only the given instances. Rather, the expectation is that the created schema is somewhat more general than the tightest fit over the samples. A good example of this are cardinalities: Sample instances will always describe a finite number of instances, even if there is no upper limit for the cardinality of an attribute. Schema creation algorithms which receive incremental feedback from users encode such expectations in heuristics [GNA99, BSG+04] in the hope that wrong guesses will be corrected by feedback. Some less quantitative quality measures are discussed in section 7.2.4 when the quality of the schema that is generated in UCSCP is assessed.

Phase III: Interaction with the Domain Expert

In the third phase, the user has three opportunities to influence the created schema: by answering questions from the system that arose in phase II., by working on the sample instances which are the basis for the created schema, and by modifying the schema directly. Also compare the feedback loops in figure 7.11.

Answering questions from the system about schema construction has a direct effect on the

schema and can influence multiple classes at a time (as the same problem, e.g., an ambiguous naming, might occur more than once). The system reacts by adjusting the schema according to the given hints.

For each class the user can inspect the instances which back it. Any undesired classifications revealed in this inspection can be fixed on an individual basis by modifying the affected instances. When all desired modifications have been made, phase II. is re-executed to yield the corresponding changes on the schema.

Modifications on individual instances can, however, be cumbersome especially if the problem affects multiple expressions. Direct manipulation of the schema is therefore also possible. When the user makes modifications on schema level, the system modifies the corresponding expressions accordingly.

A common case in which modification on schema level is used are expressions which have been collected in a common class because they are accidentially very similar in structure but unrelated semantically, figure 7.12 gives an example. Such a situation is likely to occur in parts of the schema which are only of minor interest in the application domain and therefore not richly modeled. The user can explicitly divide the expressions of the invalid class into several groups, for example along distinguishing semantic types of the expressions. This is called a *split* operation. The split operation does not require modifications to expressions, instead, it introduces explicit typing information for some expressions.

In the other direction, an explicit *merge* operation is also possible. Two classes that the user deems identical, but that technically are not, are merged in this operation. The differences between classes can be in attribute names, their types, or even in the structure of the whole class. In the first case, the user is required to equate attributes from both classes in a simple one-on-one mapping. The system modifies the abstractions on the respective instances accordingly to alert the user to the consequences of this merge. If the user chooses to modify the type of an attribute in order to merge two classes, this modification can be to a more general type (all expressions remain well-typed due to the substitutability of specific types for general ones) or to a more specific type. In the latter case the user has to supply proof that is type modification is valid by acertaining the more specific type for the applied expressions. This requires the modification of those expressions. Introductions of new attributes cause the system to introduce corresponding abstractions on the affected expressions. These expressions are then only well-typed in AE_C after these abstractions have been met with applications. Thus, extensional proof is required for new attributes. The modification of an attribute's type to an unrelated type is treated as the introduction of a new attribute.

Human-computer interaction can be understood as the reaction of the human user to visualization on screen by the computer [BCM99]. Possible reactions of the user have been described above, leaving the visualization to trigger those reactions. The visualization's task is to inform the user of the schema that was created from the supplied instances. An obvious possibility is therefore a graphical representation of the Asset classes similarily to UML class diagrams for example in the notation used in figure 7.12 and chapter 6. The domain expert can make all the modifications to the schema in this visualization. As it allows rather direct interaction, this visualization should certainly be available in a system which runs the schema creation process. A survey of visualization techniques useful in ontology development can be found in [Ng00]. Some of the techniques might also be useful in building a sophisticated visualization that helps the user in understanding large amounts of expressions and the corresponding classes. One such technique are landscapes that provide an overview of the closeness of elements.

However, it should also be recognized that semi-formal diagrams are inappropriate for many domain experts. To understand them, the user has to be familiar with modeling concepts such as inheritance. This is in practice often not the case. In an application project (see section 7.2.1 and chapter 8) class diagrams were found to be a tool largely useless for communication with domain experts as their semantics were not understood even towards the end of the project. Others [SM99] report similar experiences.

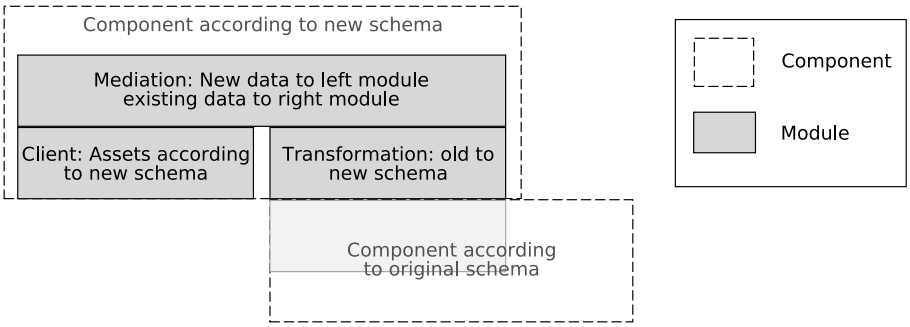


Figure 7.13: *Setup of a component for schema evolution. Instances according to the original schema are transformed to the new schema. New instances are stored in a dedicated client module. Access to both is mediated such that an application based on the mediation module only sees instances according to the new schema.*

What was, however, a suitable means for discussion was the user interface generated from the conceptual schema. It was only after the availability of this interface that the consequences of modeling decisions were understood and some decisions could be identified as ill-founded. Accordingly, the schema creation process can communicate with the user via prototypical user interfaces. This idea is also pursued in [PMM05] where a user interface prototype is created directly from the model. Prototype systems are used in software development in general not only to provide feedback to the user long before an initial version of the system has been built, but also to motivate users to participate in development [Rup04, 3.3.5].

By means of dynamic generation of CCMSs, prototypical user interfaces can quickly be provided to the user. To this end, a small CCMS is generated which is composed of two modules: a server module providing the user interface, and a client module holding the data that the schema was created from—converted Asset instances based on the sample expressions. This system is generated from the current state of the schema and therefore always reflects the user's latest changes. This allows quick evaluation of the feasibility of these changes.

The prototype system is *read-only* with respect to the schema. While new versions of the prototype are available upon each schema modification, allowing the user to make schema modifications—such as the introduction of new attributes—directly in the prototype again causes the one-instance-problem described in section 7.2.1 above. Assume that, e.g., the prototype is displaying a web page that shows a detailed view of an instance. A request for an additional attribute on this page is likely to be founded in this instance only. Conversely, checking many pages after a new attribute has been introduced by the process allows the users to evaluate the attribute's feasibility. Therefore, the prototype is used for presentation only.

Phase IV: Creation of New Conceptual Content Management System

When the domain expert is satisfied with the created schema and the presented prototype, the process enters the final phase IV. This phase is again fully automatic and makes heavy use of dynamic system generation. The system that is compiled from the schema this time is a full system not a simple prototype built from two modules. The structure of the full system is based on a component configuration appropriate for the task at hand (see section 2.1.4 on component configurations). If the process resulted in a modified schema, the component configuration is augmented with the modules to accomodate this new schema. This is done with the schema evolution pattern [SBS05] shown in figure 7.13 which lifts the module interface to the new schema. On top of the mediation module, additional modules reside to realize application-specific functionality such as user interfaces or distribution.

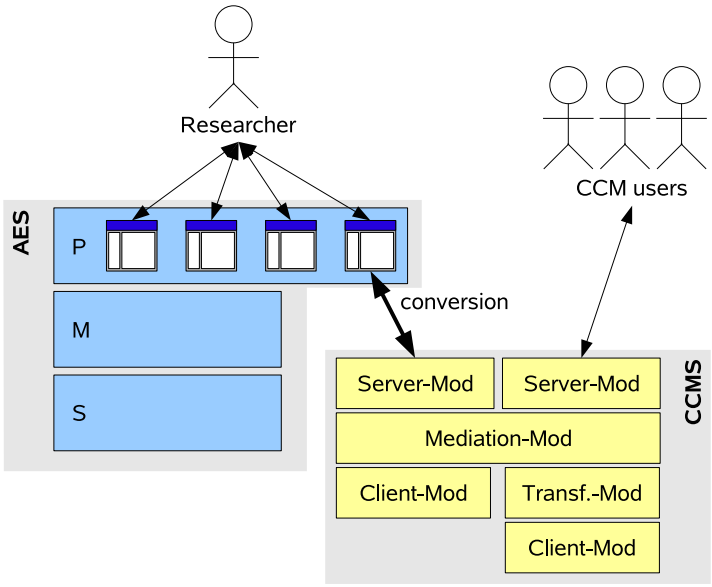


Figure 7.14: *Combination of AES and CCMS to support open and dynamic schema evolution in CCMS. Researcher models domain in Asset Expressions in the AES with complete freedom. Insights are carried over into the CCMS for all users by means of the UCSCP.*

7.2.3 System Integration

The execution of the UCSCP can result in a new or updated CCMS. At this point the lifecycle of the involved AES that was used to support the UCSCP can end. However, this need not be the case. The AES can continously exist besides the CMS as is shown in figure 7.14. The AES provides its users with a view on the information in terms of Asset Expressions. Available expressions can either be stored in the AES directly (through the means described in chapter 5) or be loaded from the CCMS. In the latter case, Asset instances need to be converted into Asset Expressions by the conversion rules provided in section 6.2. In the AES users can work with Asset Expressions that include the instances in the CCMS. The users of the CCMS work with Asset instances according to the schema that was developed in the last iteration of the UCSCP. For tighter integration of AES and CCMS the introduction of the AES as a module of the CCMS is an options (see section 2.1.4 on modules). However, the exposure of the full AES functionality would require significant additions to the module interface.

The combination of AES and CCMS shown in figure 7.14 can be used to several effects:

1. *To run the UCSCP.* In this case, it is used in phases I and IV of the UCSCP. At the beginning of the process, Asset instances can be imported into the AES to serve as a basis for personal modifications. At the end of the process, Asset Expressions are converted into Asset instances and used to populate the new system.

2. *Individual extensions.* Users can employ an AES that is connected to a CCMS to extend instances existing in the CCMS on an individual basis. Assets in the CCMS are shared among a group of users, each of whom have their own AES for extensions. This use case also appears in CCMSs in a very similar form: By means of instance personalization, users in the CCMS are able to modify Asset instances. However, those modifications are not as flexible as in the AES as they need to adhere to an—albeit personalizable—schema. If an AES is employed for personal extensions to Assets, the UCSCP can be

used to create a schema matching these extensions. This schema is personal to the user and allows for CCM-style schema personalization (see chapter 2.1.4).

3. *System evolution*. This scenario assumes most users work in the CCMS, either with a stable schema or with the personalization means available there. Some users work in the AES to explore new descriptions possibilities there to find ways to more adequately capture the application domain. Once significant progress has been made in the descriptions in the AES, the UCSCP is run. It results in a new schema for the CCMS that accounts for the new description structures developed in the AES. The enhanced instances from the AES might also be carried over to the CCMS.

7.2.4 Quality of the Created Schema

Fully automatic assessment of conceptual schemata is a difficult task in general. Several approaches to fully quantifiable metrics have been proposed [AS83], the drawbacks of which were discussed above. Frameworks to determine conceptual schema quality (see [Moo05] for a survey) also employ non-quantifiable criteria. Such a framework can use a quality measure composed of three dimensions [CACW02]: Specification, usage, and implementation. These dimensions are illustrated in figure 7.15. The conceptual schema created from Asset Expressions by the process presented above can be evaluated according to these dimensions. To this end, each dimension is broken down into several criteria [CACW02]:

1. *Specification*: Specification is concerned with how well the domain requirements can be understood from the conceptual schema. It is broken down into four criteria:

   - *Legibility*: The legibility of a schema is determined by its clarity and its minimality. Clarity is purely aesthetic as it is based on the graphical representation of the schema. Any layout algorithm used to create a presentation of the generated schema is likely to achieve results inferior to manual layouting. Minimality measures the redundancy of the schema. If the Asset Expressions used consistent naming, the generated schema contains little or no redundancy. However, if naming is diverse, the UCSCP will not collapse structurally similar classes automatically. User input is then necessary in phase III to reduce redundancy.
   - *Expressiveness*: The expressiveness of a schema can be broken down into concept and schema expressiveness. The former measures how well the concepts in the schema can be used to describe the real world. This measure is directly dependent on the quality of the Asset Expressions which were used to create the schema. If these expressions provided the necessary information on the domain, the conceptual schema is able to model this information. Schema expressiveness considers the schema as a whole. Generally speaking, a schema with more concepts can be more expressive, for example by providing refined sub-concepts where a simple schema might only use a single concept. The automatically constructed schema has an inheritance hierarchy that fully reflects the diversity of the underlying Asset Expressions. The expressiveness is therefore high.
   - *Simplicity*: A proposed measure [GJP00, CACW02] is the relative number of relationships (including inheritance) over the number of classes. This is highly dependent on the actual schema and therefore in general not measurable a-priori. However, the sample expressions that are used to generate the schema, are likely to contain some structural noise—in the sense that similar expressions are modeled slightly differently—which can lead to a high number of subclasses that differ little from their parent. When computing the inheritance hierarchy, the *Inheritance Orphan* (page 131) is designed to deal with cases of structural noise.

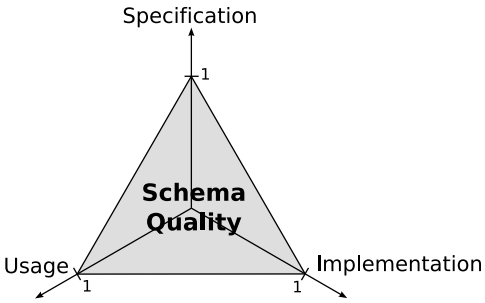


Figure 7.15: *Dimensions of conceptual schema quality [CACW02].*

- *Correctness*: Correctness refers to syntactic (correct specification of classes) or semantic properties (use of classes according to definition). The former is the case as classes are created automatically. Structurally coinciding classes with different meaning can lead to semantic incorrectness of the automatically generated schema, figure 7.12 gives an example. Such errors need to be corrected by user intervention.

2. *Usage*: The usage dimension includes the schema's coverage of relevant features of the application domain and the overall understandability of the schema by human readers. The quality of the conceptual schema can be assessed according to how well it matches the user's idea of the application domain. Two criteria are used for this:

   - *Completeness*: A schema that represents all relevant aspects of the application domain is said to be complete. It is assumed here that those aspects have been modeled in Asset Expressions by the domain expert. Therefore the schema is complete with respect to the specifications of the user. If the Asset Expressions created by the user to describe the domain have modeling deficiencies—e.g., omit certain aspects or describe them too weakly—which will cause the respective parts of the schema to exhibit the same weakness. Completeness is therefore highly dependent on the quality of the sample expressions.
   - *Understandability*: The ease with which the user can interpret the schema is of great importance to its validation. The choice of meaningful names for classes and attributes plays a crucial role in this. On the one hand, attribute names in the generated schema are carried over from the Asset Expressions in which the name was chosen by the user. It can therefore be assumed to be reasonable meaningful to the user. Class names on the other hand are invented by the system, if possible based on the names of a suitable semantic type. If no such semantic type can be found, a generic name is used. The choice of class names can thus be rather meaningless, requiring additional input from the user.

3. *Implementation*: The effort required for the first implementation of the schema in an information system and the effort for maintaining this system upon schema changes are summarized in the implementation dimension:

   - *Implementability*: Refers to the amount of work that is necessary to create an information system based on this schema. For an Asset schema, this work is done by the compiler framework, no developer is needed. All additional manual work is not schema-related, the implementation effort can therefore be considered to be roughly independent of the schema. Excessive schema sizes—caused by, e.g., a too fine modeling of classes with few differences—will of course lead to much larger generated systems. However, given common execution times of the compiler framework, any practical schema is unlikely to cause problems.

- *Maintainability*: Maintainability measures the ease with wich the conceptual schema can evolve. Schema evolution is an important use case of Conceptual Content Management Systems which is supported well by compiler framework and system architecture, see section 2.1.4. Important questions such as model cohesion and coexistance [CACW02] are addressed by the Conceptual Content Management approach.

Schema generation creates a schema that is of high quality with respect to its coverage of the application domain as described by the user. Further advantages are the automatic provision of schema integrety and syntactic correctness. Problems arise in areas where creativity is necessary: the choice of appropriate base expressions or the naming of classes. These must therefore be left to the user. Due to the use of CCM's system generation approach, implementation complexity does not burden the user. Larger schemata with richer models can thus easily be tolerated.

Chapter 8

Application Example

This chapter evaluates the Asset Expression approach in the context of an application project. Many of the insights into entity descriptions were derived from this project. The application project described in this chapter was carried out in parallel to the development of Asset Expressions. Therefore, important concepts of Asset Expressions appeared in the project. Below it will be shown how some of the deficiencies of the developed information system that was created in the project can be remedied by use of Asset Expressions.

Because of the chronology of events that the project was not carried out with Asset Expressions in place beforehand. This chapter therefore demonstrates how the domain is fully modeled with Asset Expressions and how this affects the produced information system.

In the early phases of the project the only concepts from Asset Expressions that were already present were content components and their connection to description attributes. Among the project members there was a demand for what was called "free-form" descriptions of entities. However, it was quickly acknowledged by all participants that completely free-form entity descriptions were unlikely to yield satisfactory results in a collaborative project. Instead it was decided to go ahead with what was available at the time: structured descriptions as a basis for a Conceptual Content Management System (section 2.1.4). In fear of demanding unapplicable information on entities, no mandatory attributes were introduced, only a few received "recommended" status. A prototype that connects content components with conceptual attributes was also built [Uri05]. The prototype works on Asset instances and relates attributes to parts of the content of the instance. It can handle images and plain text content. To implement the connection of attributes and content components, the prototype uses an explicit meta-model for content components as described in section 6.2.2.

A central part of the project is its classification taxonomy. For practical purposes this taxonomy also contains instances of some of the concepts it provides. The semantic types presented in this chapter are concepts from this taxonomy. They were obtained by removing all instances from the taxonomy and introducing a few additional types to better structure the existing types.

8.1 The Application Domain

The name of the project is "Geschichte der Kunstgeschichte im Nationalsozialismus", which translates to "history of art history during the national socialism" and is abbreviated GKNS. It is a collaboration of the art history institutes of the universities of Berlin (HU), Bonn, Hamburg, and Munich as well as the Software Systems Institute at the Technical University of Hamburg-Harburg. For more information on this collaboration and on the project itself please refer to its homepage[1] and the web interface of the Conceptual Content Management System,

[1] http://www.welib.de/gkns

which can be found online[2]. The system is generated from a conceptual schema, which can be obtained from Asset Expressions by means of the UCSCP.

The project collects large amounts of documents dealing with art history in Germany in a timeframe between 1930 to 1950. The aim of the project is to provide an integrated set of resources to enable future research by art historians on the topic. This set is difficult to obtain by traditional means—i.e., research in archives—because the documents are scattered over many archives across Europe. To facilitate the coherent interpretation of these documents, the project's aim is to collect documents electronically and to provide research facilities on this collection. Therefore, documents need to not only be scanned but also described conceptually.

Future research based on the collected documents calls for the digital representations to be as objective as possible such that research is not biased later. Measures that encourage objectivity as well as collaboration between users are discussed in section 8.4.

While the entities of central importance are the documents themselves, adjacent entities such as persons, institutions, and geographic locations are also included in the application domain. The purpose of the expressions modeling these entities is mainly to interconnect documents and not to provide a complete model of the respective entity.

8.2 Asset Expressions for Domain Entities

The expressions representing documents have the richest model in the GKNS domain. Figure 8.1 shows an example of such a document. Two components that are described via abstractions are the persons occurring in the document. As done on the left-hand side of the figure, the applied expressions can model the persons to the extent necessary in the context of the document. The other two explanations deal with date and place of the document. In the case of figure 8.1 it might be expected that a human with some knowledge in the application domain would know where the city of Hamburg is, making its explicit mention superfluous. It is, however, difficult to draw a precise line between those geographic locations for which this is true and those for which it is not. It is therefore reasonable to always make the geographic location explicit regardless of its popularity. Figure 8.1 does not reference other expressions to illustrate how a rich network of contents can be used to create semantic neighbourhoods between related contents.

Figure 8.2(b) shows an expression which describes a document by mostly using already defined expressions. One of the predefined expressions is shown in figure 8.2(a). Besides referencing expressions directly, users also have the option to create a new, richer description of the entity which better suits the context of the explanation. Suppose that in the example of figure 8.2(b) the expression describing the art history institute ($\mathsf{KHIBonn}_{orig}$) could be more helpful to the explanation if it also included the employees:

$$\mathsf{KHIBonn}_2 := (\lambda \text{mitarbeiter: } \mathsf{WissenschaftlichesPersonal}^*.\mathsf{KHIBonn}_{orig})\ \{\mathsf{AnnaWarburg}, \mathsf{AlfredStange}\}$$

If this description is used in explanations in place of the original one, viewers benefit from the added information but can also still connect the overall expression to the original model for the institute. To retrieve all documents which use the institute in an explanation, one might for example pose the following query:

$$/^*[//\mathsf{application}/\mathsf{operand}//^*=\mathsf{KHIBonn}_{orig}]$$

Expression references can be expanded by a presentation system (chapter 5) into large networks such as the expression in figure 8.1. Modeling each entity in its own expression instead of putting everything into one large network has the advantages of making entity boundaries

[2] http://www.welib.de/gknsapp/showlogin.do

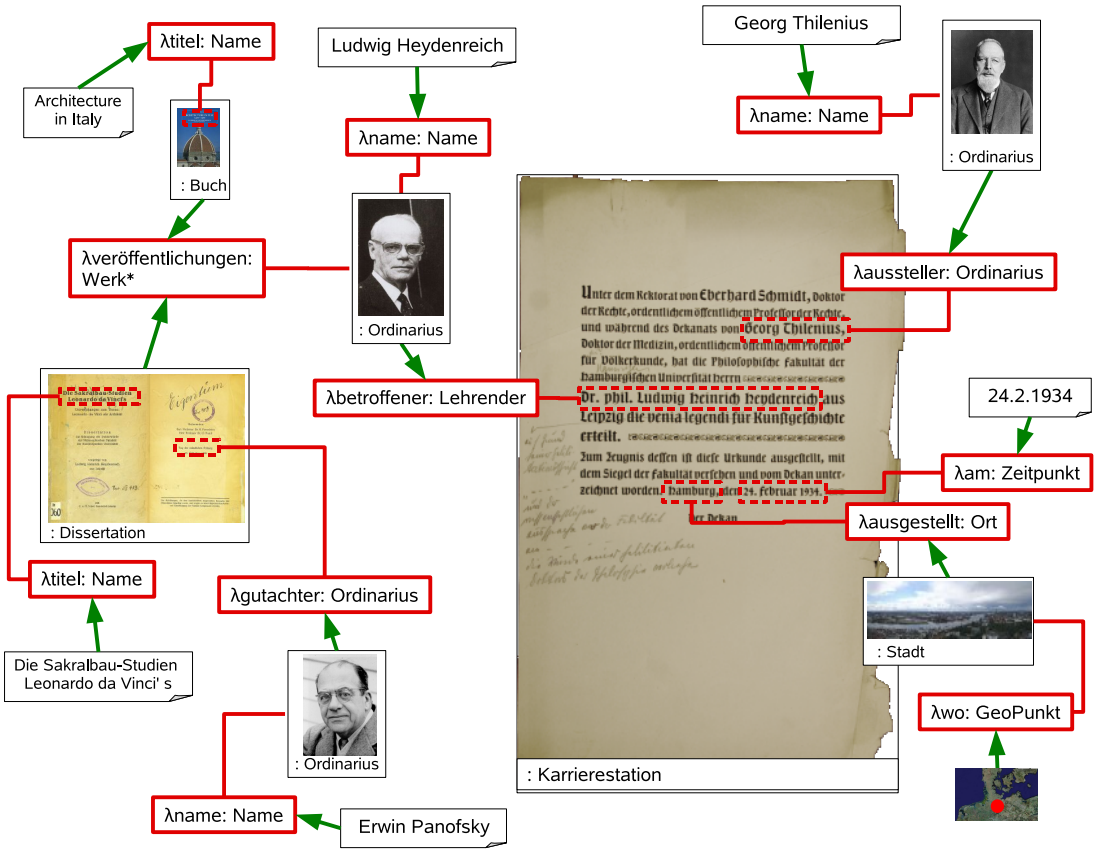


Figure 8.1: *Venia Legeni for Ludwig Heydenreich. Semantic types for plain contents are omitted.*

explicit, providing hints for the simplification and enrichment of expressions (by collapsion of applied expressions and references to other expressions in explanations), and reducing the cognitive load on users familiar with the application domain (by not requiring them to re-detect large patterns of recurring expressions).

8.3 Semantic Types

The hierarchy of semantic types was defined in a joint effort in the GKNS project. While some additions were necessary during the course of the project, there were no personal variants of types for particular users. The hierarchy is a combination of types from several domains, some of which are specific to the project (institutions, roles, etc.) while others are not (e.g., time and topology). Figure 8.3 shows an overview of all domains but omits many types, only listing some examples from each domain.

Jointly developing the semantic types is important to involve users who are not at ease with hierarchical relationships. This problem is not special to the GKNS project and has also been observed by others [GW02]. Potential misuses of the sub-type relationship weaken its "is-a" semantic by including all sorts of relationships between semantic types—such as "related-to" or "part-of"—where an "is-a" relation is required by the substitutability of Asset Expressions. Involving modeling experts is a common remedy to this issue [GW02], the experiences in the

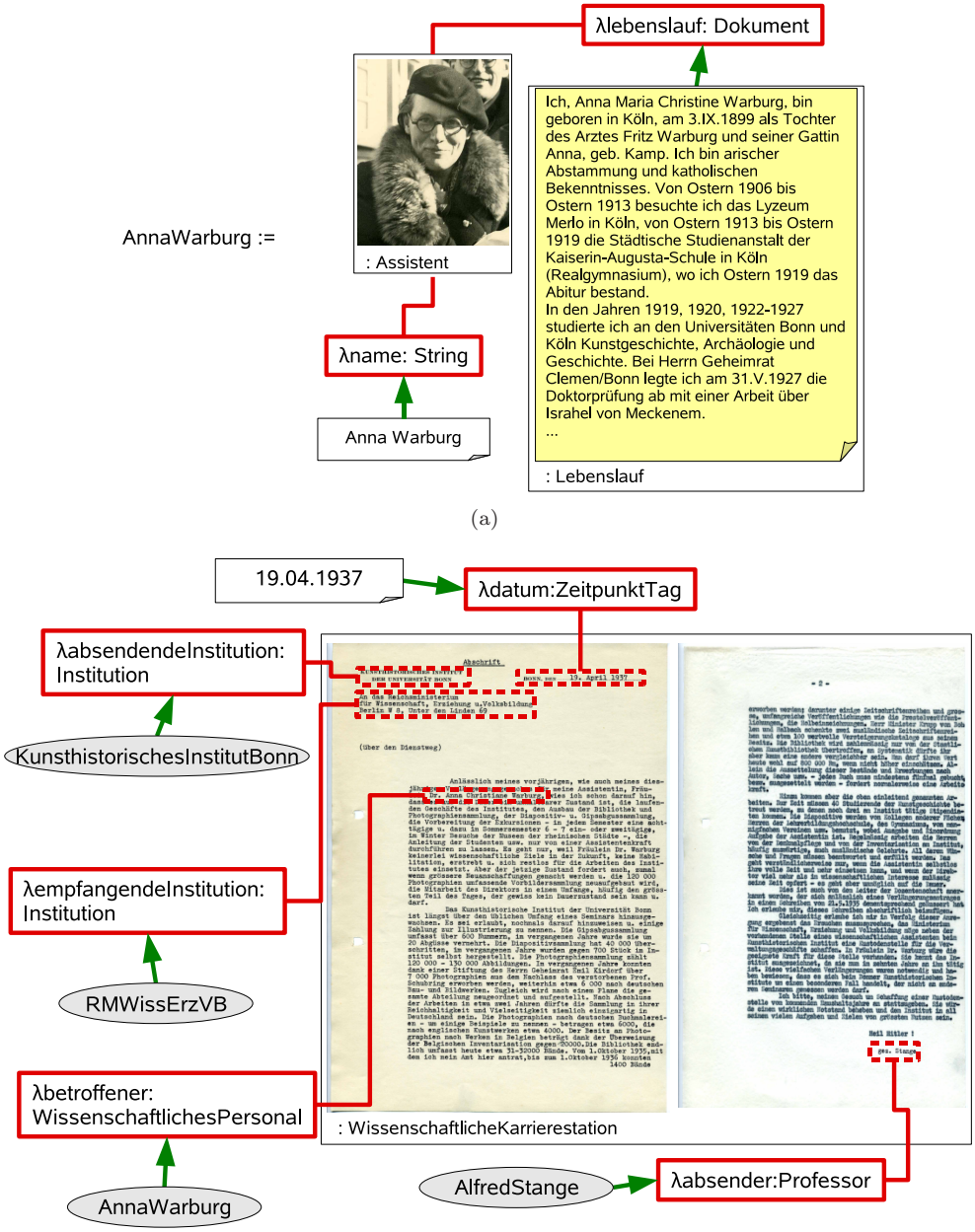


Figure 8.2: *Definitions of recurring entities are used to describe documents. Semantic types for plain content are omitted.*

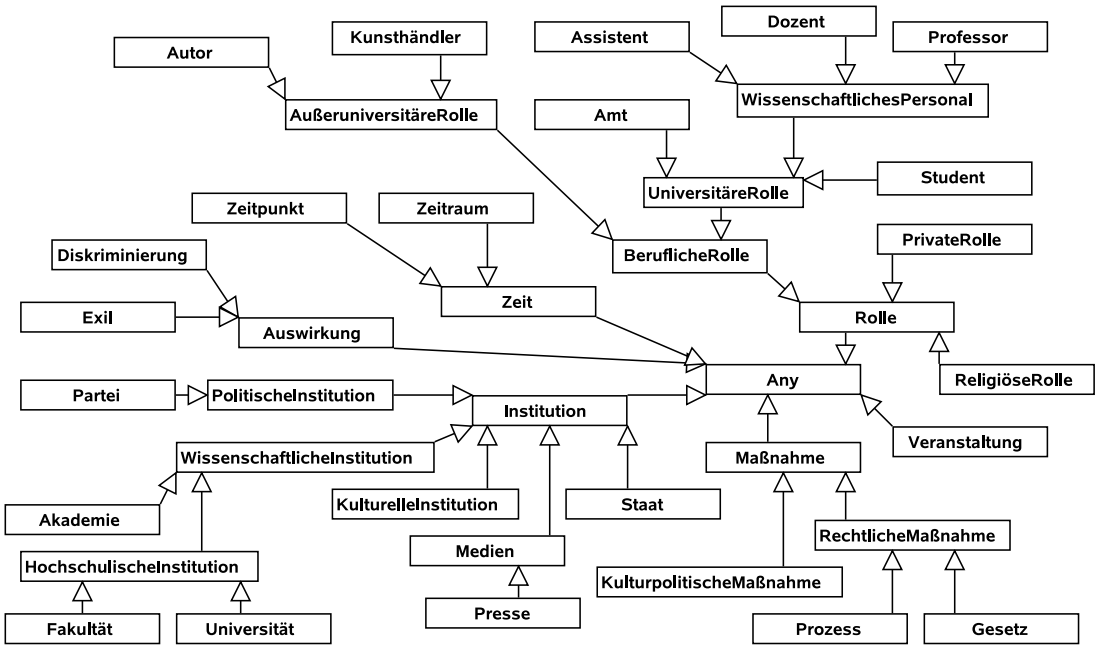


Figure 8.3: *Summary of semantic types from GKNS. The top-most semantic type* Any *is located near the center of the figure.*

GKNS project suggest that some additional training for domain experts can be sufficient if a very structured style of discussion is adopted.

8.4 Collaborative Creation of Expressions

A central aspect of the GKNS project is collaboration. The Asset Expressions modeling archived documents therefore had to be constructed with special emphasis on interoperability between users. To this end four means are necessary:

1. Agreed semantic types as introduced in the previous section.

2. A rather high number of traits to deal with recurring needs with respect to the structure of descriptions.

3. Editorial guidelines which detail many aspects of expression creation such as when to use which trait or semantic type, as well as how to deal with uncertain or unknown data.

4. Common descriptions for secondary entities. These entities are modeled because they are related to the documents of interest. Examples include persons who might occur as sender of letters as well as authors of documents or institutions in similar positions.

Below some of the traits for the GKNS domain are shown. While the semantic types stem from different domains, their namespaces are omitted for brevity as the type names are unique across domains. A brief break-down of the domains was discussed in the previous section. The semantic type prescribed by the traits is usually Any. The traits are constructed to reflect different kinds of documents which occur in the archives. The semantic types of particular expressions however, are determined by the contents of the respective documents. A

lax prescription of semantics types in the traits therefore ensures the applicability of the traits in as many scenarios as possible.

Traits are organized in a hierarchy for two purposes. The first is to factor out similarities between traits to keep the definitions more concise. The second is to guide users in choosing a trait by navigating the hierarchy top-down until they arrive at the most specific trait which meets their requirements.

trait GKNSDokument **refines** EmptyTrait **of** Any **with** λdatum: Zeitpunkt.λstandort: Signatur

This trait captures common abstractions of archival documents. Any document described in the GKNS project needs to supply this information in order to be usable by allowing its basic placement in time and the discovery of the original from the archive by means of the archive signature.

In a project which deals with the history of a scientific discipline, publications are of great importance. A small hierarchy of traits for all publications captures this, in particular to suggest some details of expressions modeling scientific publications.

trait Veroeffentlichung **refines** GKNSDokument **of** Any **with** λautor: Rolle.λjahr: Zeitpunkt.
λverlag: Verlag

trait Habilitationsschrift **refines** WissenschaftlicheVeroeffentlichung **of** Dokument **with** λautor:
Rolle.λtitel: String.λdatum: Zeitpunkt.λgutachter: Professer.λgutachten: Gutachten

trait Dissertation **refines** Veroeffentlichung **of** Dokument **with** λtitel: String.λdatum: Zeitpunkt.
λgutachter: Professer.λgutachten: Gutachten

There are also semantic types for different kinds of documents which should not be confused with traits giving structure to the descriptions for these documents. In the last trait definition Dissertation the semantic type is Dokument which is unrelated to the trait GKNSDokument.

Dissertation and habilitation are also provided with traits as they are important steps in academic careers which allow interesting connections of their authors to other persons in the field. Other types of publications (e.g., journal articles, books, etc.) can of course also be richly modeled (e.g., with the journal they appeared in) but this is of less interest here as the GKNS project does not aim to build a publication database of the time.

As a reflection of processes, institutions or personal lifes, factual documents play an important role in a project which strives to enable research in a domain by providing an objective archival basis. Factual documents come in a large variety, making it helpful to acknowledge the most interesting ones with traits:

trait Sachdokument **refines** GKNSDokument **of** Any **with** λverfasser: Rolle.λbetroffener: Rolle

trait Antrag **refines** Sachdokument **of** Any **with** λsteller: Rolle.λadressat: Institution

trait Ausweis **refines** Sachdokument **of** Any **with** λhalter: PrivateRolle

trait PersonenFoto **refines** Sachdokument **of** Any **with** λfotograf: Rolle.λabgebildet: Rolle

trait Lebenslauf **refines** Sachdokument **of** Any **with** λperson: Rolle

trait Publikationsliste **refines** Sachdokument **of** Dokument **with** λautor: Rolle.λeintraege: Any*

Providing a minimal set of abstraction is of particular importance for correspondence documents such as letters. These documents make some sense on their own but their full influence can only be assessed in the context of the whole chain of correspondence. The necessary information is required by the abstractions in the following trait:

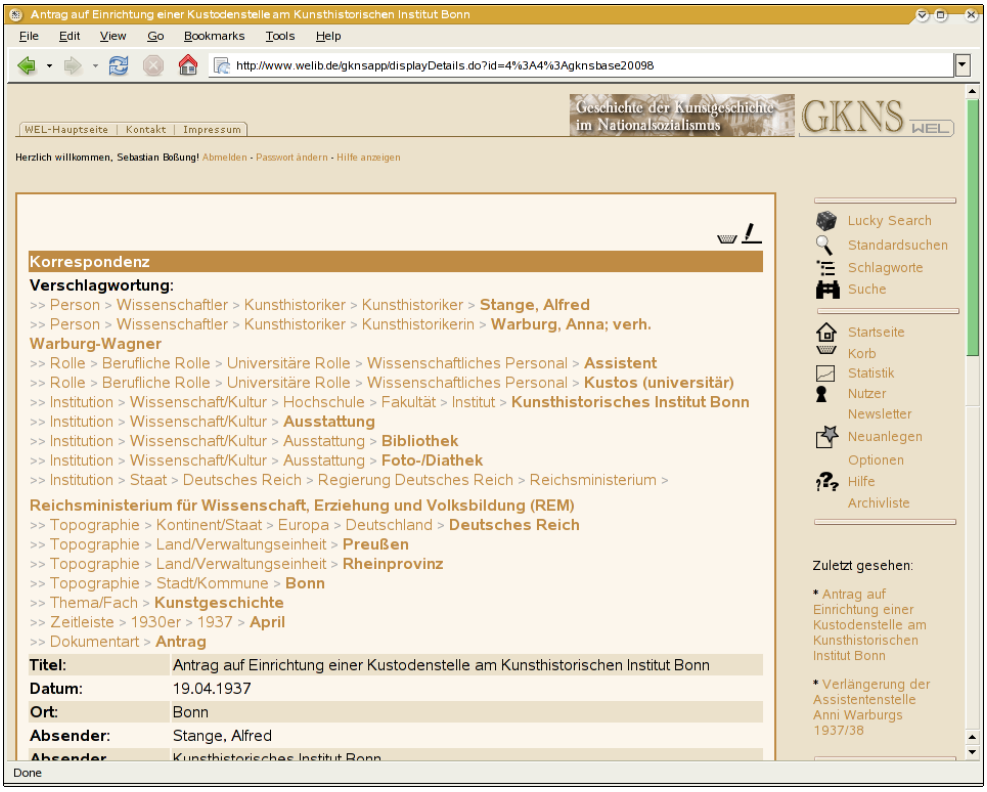


Figure 8.4: *Screenshot of the GKNS user interface. This user interface is also provided by the prototypes during the schema construction process and allows users to directly assess the implications of changes.*

trait Korrespondenz **refines** GKNSDokument **of** Any **with** λabsender: Rolle.λempfaenger: Rolle.λabsendendeInstitution: Institution.λempfangendeInstitution: Institution

The majority of documents in the project can be described by expressions based on this trait. Additional traits include expert opinions (*Gutachten*), lecture notes (*Mitschrift*), and laws (*GesetzErlassBestimmung*). These are not as common as correspondences.

trait Gutachten **refines** GKNSDokument **of** Any **with** λautor: BeruflicheRolle.λbegutachtet: Any

trait Mitschrift **refines** GKNSDokument **of** Any **with** λautor: Rolle.λthema: Veranstaltung

trait GesetzErlassBestimmung **refines** GKNSDokument **of** Any **with** λerlasser: Institution. λunterzeichner: Rolle*.λveroeffentlichungsdatum: Zeitpunkt.λwirksamkeit: Zeitpunkt

The traits provide a minimal description framework for most expressions that are based on documents.

8.5 Conceptual Schema and Information System

In the steps described in the previous sections, great emphasis was put on collaborative features. This has the effect that expressions which describe similar entities share a common kernel of explanations (due to the trait employed to create them) and are typed from a common system

Figure 8.5: *Screenshot of the data entry tool from the GKNS project. The photograph in the newspaper article on the right contains two content components.*

of semantic types. These two features of course greatly facilitate the creation of a conceptual schema with the schema creation process from section 7.2. It is acknowledged that some work for the creation of the schema has already been done by creating the traits. However, the need for collaboration rarely comes as a suprise and a minimal joint effort of modeling and domain experts to create the traits can reasonably be assumed at the beginning of the project.

A conceptual schema for the GKNS domain can be created semi-automatically from Asset Expressions such as the examples shown earlier in this chapter. The classes in this schema are from three groups. The first group are those classes describing archived documents. This group is richly modeled as various details are of interest in the application domain. The second group are classes which—while specific to the application domain—are not of interest by themselves but are mainly used to describe documents. Examples from this group are several types of persons as well as different institutions which produce or are subject of documents. The semantic types of these expressions are rather diverse (figure 8.3 shows parts of the hierarchies for Rolle and Institution). The third group are classes that are not specific to the application. They provide information on time and locations. It makes sense to import this third group from existing schemata to allow for some interoperability with other systems by this minimal common model. There are a several proposals which deal with time [PSZ99, HP06] and geography [PSZ99, OGC04].

Compromises between the richness and the complexity of the conceptual schema need to be made, especially in the first group. The classes in the second group tend very much towards simplicity as their instances serve as connections between documents. Elaborate models of persons or institutions are not of interest in the project. In fact, the manually developed schema of the GKNS project took an extremely simplistic approach to the second group: persons and institutions are only modeled by their name. They were even introduced into the classifier taxonomy to create a user experience very similar to a previous system.

A system configuration was set up to generate a CCMS from the created conceptual model. This is clearly a task for a systems expert and beyond the scope of the domain expert. Figure 8.4 shows the web-based user interface of this system. The document displayed is the one modeled as an Asset Expression in figure 8.2(b). The setup of the GKNS system is shown in figure 8.6,

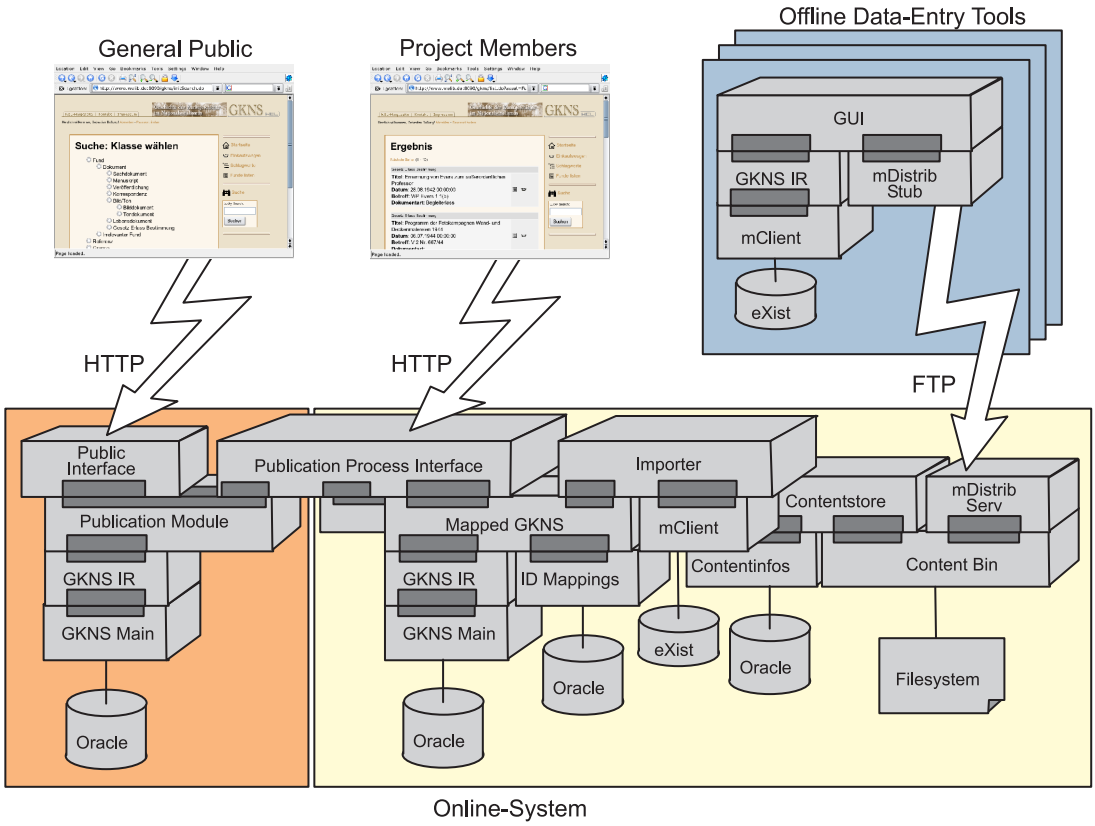


Figure 8.6: *The architecture of the GKNS system. Several of the modules are in fact macros of module patterns, omitting several mediation modules.*

some additional information on the system can be found in [BSHS06]. Figure 8.5 shows the offline data-entry tool for the project. The system provides the functionality expected from a modern distributed information system. It includes a data entry tool with ansynchronous replication to the central system, a staged editorial process for quality assurance, as well as user management and access control. Not visible in the figure are non-functional features such as efficient search over very large datasets.

Prototypical systems created as feedback to the user during the schema construction process (see section 7.2.2) also have the interface of the full system and show the same data, as the latter comes from the Asset Expressions. Compared to the module configuration of the productive system shown in figure 8.6, the prototype setup is much simpler with only two modules. This makes is feasible to create several different prototypes successively as both system generation and startup time are suitably short.

8.6 Evaluation

As illustrated above, Asset Expressions can be used to model the application domain, with the greatest benefit if the description of entities can be based on a medial representation. Networks of expressions put different entities—each is represented by medial content—in visual proximity, allowing users to quickly grasp their dependencies with minimal navigation through expressions (compare figure 8.1). Such networks are therefore valuable to the domain expert as well as the casual user. To reach larger audiences of casual users it is however necessary to reduce

entrance barriers as far as possible. This can be achieved by providing a user interface of a familiar kind, today mostly web-based interfaces.

Issues encountered in applying the Asset Expression approach to the GKNS project are mostly standard ones. Properly setting up the system of semantic types is a non-trivial task which cannot be carried out successfully by any domain expert alone in a project the size of GKNS. On the positive side, the type system does not require frequent changes. In fact, when changes occur, they need to be made after a concensus has been reached in the whole group to avoid fragmentation. Use of a type hierarchy with "is-a" relationships can take domain experts some time to become aquainted with. The lack of structural consequences of semantic types on the typed expression certainly facilitates this understanding.

In the creation of the semantic types, problems occurred which can be observed in similar projects regardless of modeling paradigm. One is the proper modeling and use of persons in roles, another the description of topological locations which change over time (e.g., countries, cities belonging to countries, etc.). The issue of roles can be addressed in Asset Expressions by introducing different semantic types for each role that is of interest in the application domain (see figure 8.3 for the GKNS approach to this). The same physical person can then be modeled in different roles by creating several role expression in which (a) the same content is typed with the respective semantic types, and (b) the explanations on the expression reflect the aspects about the person specific to this role. The expressions are then used in the appropriate places—which, e.g., require the person in a personal or professional setting in the case of GKNS. Through the identical content the role expressions are connected, creating a multi-factet model of the person that can be interesting on its own.

Topological issues in GKNS were manifold due to the the time the project is concerned with: Countries changed shape or name, they came into being or vanished; accordingly cities changed names as well and moved from one country to another. The adopted solution to this diversity problem was to model each country independently of its extent—the project's interest in countries is mainly political not geographical—with different expressions for new countries in a political sense, while cities are modeled as one continous expression, possible providing different names. In combination this means that cities must not be connected to a country in general as this connection would need to change over time, therefore the model of cities is not time dependent. Such connections can, however, be made on the basis of individual expressions, for example by creating a richer model of a city before using it in explanations (analogously to the modeling of the institute in section 8.2).

Depending on the background and size of the user community, the freedom of modeling with Asset Expressions can be very beneficial or can cause some problems of coordination. When creating expressions, AESs offer help to users in a variety of forms (traits, components, visual notation) but enforce only few restrictions on the user. This allows users to choose the model most appropriate to their needs. However, if users collaborate to create a set of expressions that coherently models an application domain, they need to use means outside the system to ensure coherence. It was found in the GKNS project that the domain experts were very much at ease with ensuring coherence by social means (e.g., editorial processes and guidelines). In fact, most restrictions imposed by the structured information system were fiercely debated, as it was felt that they were inappropriate to some of the entities.

Chapter 9

Summary and Discussion

This chapter summarizes the contributions of this thesis. It then relates the presented approach to other ideas in the same or overlapping fields and finally presents possible directions of future research on the topics.

9.1 Contributions

Marshall McLuhan has coined the phrase of "media [as] the extensions of man" [McL94]. A medium in the sense of McLuhan is a means that augments a human capability. Humans can speak to others face-to-face in a conversation for example. Telephones augment this ability by allowing the conversation to take place over long distances. Recording technologies make similar achievements for displacements in time. Asset Expressions are a form of recording. Their purpose is to enable humans to communicate the meaning of medial content. They do so by combining medial content into larger networks of explanations. To provide and handle such explanations this thesis presents three contributions: (1) languages for defining and working with Asset Expressions, (2) systems supporting the lifecycle of Asset Expressions, and (3) pragmatics showing how Asset Expressions can be created efficiently and how further conclusions can be drawn from them.

Asset Expressions are a modeling paradigm for entities of the real world. The key notion of these models is the combination of medial representation of the entity with a conceptual explanation, which again uses content. This meets the requirements of Ernst Cassirer (see section 1.1) who stated that a concept cannot exist without extensional backing just as the entity providing this extensional backing cannot be recognized without the corresponding concept.

Three design principles have great influence on the Asset Expression approach: "Ease of use", "formality to help", and "codifyable understanding" (figure 9.1). The first refers to a domain expert's use of the approach to create descriptions of entities. This can be done with relative ease due to a simple formalism. The second is formality to help the user to create expressions. It comes in the form of semantic types and traits. The goal is to maximize the profit for the user in the trade-off between formality and ease-of-use. The third refers to the ability of the formalism to accommodate explicit (*codified*) information about the application domain that is of value to a machine. Examples of codified information are semantic types and the schema creation process.

Chapters 3 and 4 explain the Asset Expression language. It can be used to describe entities by providing descriptions of a medial representation of these entities. In their simplest form, Asset Expressions allow users to create any model without regard to its semantics. Users are free to, e.g., create non-sensical models, which have no correspondent in the application domain because they relate medial representations of entities which are—per the semantics of the application domain—unrelated. Asset Expressions do not assume the a priori presence of an

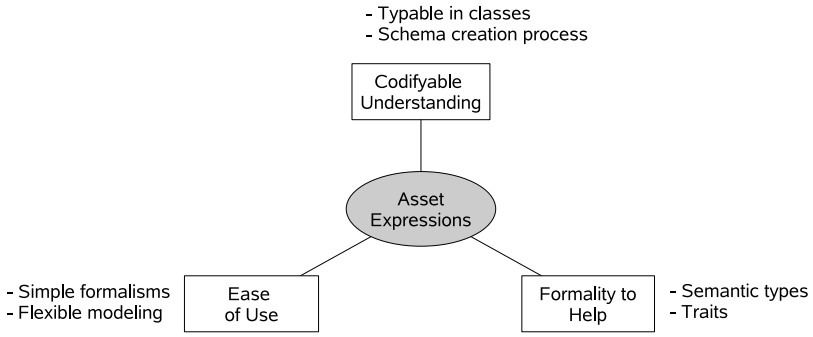


Figure 9.1: *Design principles of the Asset Expression approach*

intensional conceptualization of the application domain. As this model does not exist a priori, it cannot be used to enforce the semantics of the application domain in created expressions. In fact, such a model of the application domain can be created a posteriori in the form of a conceptual schema based on the expressions describing the domain. To inject some knowledge of the application domain into expressions, Asset Expressions are equipped with a type system (section 3.3).

Asset Expressions create a co-occurrance of explained and explaining content that is easily visualized. By means of the visual representation of expressions, this is a major enabler for domain experts who can quickly grasp the model of the application domain. Working with Asset Expressions therefore requires the direct incorporation of content representations in expression presentation. Given that the content is multimedial, this can be achieved best in dedicated, computerized systems. Such an Asset Expression System is presented in chapter 5. It allows the direct creation and manipulation of Asset Expressions as well as browsing through existing expressions. The system is also designed to let users collaborate by inspecting and referencing each other's expressions. By these means, joint models of an application domain can be built or existing domains can be combined. This is achieved by collecting strongly related expressions in a common workspace but making the borders of this workspace penetrable to allow the relation with expressions from different workspaces. Furthermore, the system serves as a basis for higher level services on Asset Expressions that can, e.g., make the creation of expressions more efficient or even provide migration paths to schema-based information systems.

The creation of Asset Expression networks is discussed in section 4.6. Collaboration of users benefits greatly from the possibility to successively create descriptions that become richer with every step. As there are no structural constraints which limit the reuse of such augmented expressions, the richer model can be used in places where the weaker model was expected. Users are therefore able to create the most appropriate explanation on a per-instance basis without hurting reuse or fragmenting the application domain by populating it with redundant expressions. Section 8.4 provides an example of this.

An important step in the creation of an Asset Expression is the choice of a semantic type for the described content. Generally speaking, the creator of the expression is free to choose any type, but might struggle to find an appropriate one in the hierarchy of semantic types for the application domain. To facilitate this, section 7.1 presents a means to discover other expressions with similar explanations. These similar expressions can provide creators of new expressions with hints on appropriate semantic types. Similar expressions are discovered by comparing the sets of types of their abstractions. This can not only be used to suggest semantic types for expressions, but also as a query facility to discover expressions that use explanations the user is interested in.

Flexible expression mechanisms allow domain experts to describe and understand the ap-

plication domain. Asset Expression Systems (AESs) support this work based on individual expressions, which can be organized, queried, or otherwise lifecycle-managed in the system. Contrary to AESs, most software systems provide their functionality not on a per-instance basis but exploit structural similarities of their data to form higher levels of abstraction and define services on these. A common example of such an abstraction are classes. The uniformity of their instances can be exploited in nearly all parts of the software system, e.g., in user interfaces, distribution, or access control. If the system is used to manage large amounts of data, such uniformity becomes a necessity, as it is otherwise impossible to efficiently perform services to the user.

In general, uniformity is not found among the expressions in an AES. However, once the application domain has been extensively modeled and is well understood, it makes sense to realize the benefits available to class-based software systems for the data in the AES. At this state of development of the domain users are likely to have understood the central structural aspects of it. These structural aspects can be captured in a conceptual schema by the schema creation process presented in section 7.2. The goal of this process is the discovery and exploitation of common structures that emerge from Asset Expressions. In traditional software development their discovery is the task of a modeling expert who closely works with a domain expert to develop these structures. In the UCSCP the AES largely takes the role of the modeling expert.

Using the presented schema construction process, the domain expert is enabled to create a conceptual schema for the application domain based on the provided Asset Expressions. Familiarity with the conceptual modeling paradigm is not assumed as the domain expert can receive feedback in the form of prototype systems. The process creates intermediate conceptual schemata, which are used to generatively provide prototype systems—most importantly the user interfaces of these systems. If the user is unsatisfied with the prototype, the schema—and through it the prototype—can be affected by either modifying the underlying Asset Expressions or taking direct action on the schema, given the user is at ease with the latter.

9.2 Comparison with Related Approaches

Approaches related to the presented work can be divided into two areas: (medial) descriptions of entities and processes to obtain conceptual schemata and the associated information systems. The former will be discussed in section 9.2.1, the latter in section 9.2.2.

9.2.1 Descriptions of Medial Content

Asset Expressions provide structured descriptions over medial content. There are several other approaches which overlap with Asset Expressions, seven of which are discussed here:

1. *MPEG-7* is largely concerned with substructuring of medial content, but it also provides narrative worlds for conceptual descriptions.
2. *MPEG-7 and ontologies* can be combined to give the narrative worlds a sound conceptual basis.
3. *Annotations* attach additional information to medial content.
4. *Markup* embeds structural information into content.
5. *Hypermedia* deals with content that can be interconnected on its own.
6. The *Semantic web* provides semantic descriptions of online resources.
7. *Structured schemata* usually do not consider medial content explicitly but provide strong conceptual descriptions.

	target audience	sematics in same paradigm	formalisms for integration	addressed media
Asset Expressions	humans (domain experts)	yes	yes (typing, components)	variety of multimedial content, extensible
MPEG-7	computers	no (but in same standard)	yes (narrative worlds)	audio and video, extensible
MPEG-7 with ontologies	computers	no	yes (narrative worlds and ontological means)	audio and video, extensible
Annotations	humans	usually not	no	wide variety, limited in particular implementations
Markup	humans/ computers	no	dependent on implementation	several, but only specialized formats
Hypermedia	humans	yes	no (some extensions available)	hypermedia documents encompass different types of content
Semantic web	computers	no	yes (limited)	many
Structured schemata	computers	n/a	no	no explicit handling of content

Table 9.1: *Comparison of approaches to interrelate content and/or capture its semantics*

These approaches will be compared to Asset Expressions, most have been described individually in chapter 2. Table 9.1 provides an overview. The comparison will be based on four criteria, which expose differences and similarities of the approaches:

1. *Target audience*: The actors who typically consume the instances or documents in the approach. Generally, these can be computers or humans, requiring means (syntax, editing facilities etc.) which are very different in nature.

2. *Modeling paradigm of semantic descriptions*: All approaches provide means to describe the modeled entity, but the paradigms used to do so are vastly different. An interesting point is whether the semantics must be described by specialized means or whether the means used to provide the content instances suffice.

3. *Formalisms for integration*: Semantic descriptions need to be related to the content representation they describe. This integration differs in the degrees of formality and standardization, for some approaches it is implementation dependent.

4. *Addressed media*: The types of medial content which are addressed by the approach. Some provide a fixed set, others are extensible, yet others can be applied to many, but concrete incarnations are limited.

The target audience of Asset Expressions are human users, more specifically domain experts. A medial representation of the described entity is explained by other instances of medial

content. This pattern is applied recursively, leading to integrated semantic descriptions. Explanations can either be over the whole of the content or over one of its components. In both cases the explaining expression is tightly integrated with the explained expression. Asset Expressions can incorporate a variety of medial contents (compare, e.g., the selectors in section 3.7) and are designed to be extensible should the need arise to explain previously unhandled kinds.

At first glance, MPEG-7 seems to be very close to Asset Expressions. It provides content components (called *segments*) and allows the description of individual components or the content as a whole. While it is mainly aimed at audio or video content, the standard provides extension points, which can be used to incorporate any kind of medial content. Content components in MPEG-7 are described in *narrative worlds*. These descriptions cannot use medial content but are confined to instances of concepts defined for the narrative world and relations between these instances, see figure 2.14 on page 37. This is an important difference between Asset Expressions and MPEG-7 descriptions of medial content. When considering the target audience, it becomes clear why MPEG-7 does not use medial content for the description of narrative worlds: The descriptions would not be understandable for a machine, which lacks the cognitive means to interpret a medial description. Therefore MPEG-7 builds a fully structured model in the narrative worlds using only defined concepts—some are provided in the standard, but the set is extensible—and machine understandable primitive literals.

The descriptions in the narrative worlds re-tell the story of the medial content, such that the machine—which in general cannot draw conclusions that seem trivial to humans from the medial content alone—can understand it to the degree necessary for the application at hand. From a human point of view, most of the given information is redundant as it can be obtained directly from the content at a glance.

Asset Expressions and MPEG-7 descriptions thus solve a similar problem: providing additional context information that enables the target audience to understand a piece of medial content to some degree. However, the means to provide this context information and the level of detail of the information are very different due to the target audience.

The narrative worlds of MPEG-7 suffer from the lack of semantic definitions of the employed concepts [HBHV04, Tro03]. This presents a problem to machine interpretation. Combining ontologies with MPEG-7 can remedy this situation as described in section 2.14. Such a combination does, however, not make the description more useful to humans. One might argue that, quite the contrary, narrative worlds become even more difficult to understand due to the combination of two paradigms.

Document annotation approaches provide additional information on parts of documents. In this basic notion they are similar to Asset Expressions. The target audiences are also similar: human users who work with the document, i.e., commonly domain experts. Annotations were discussed in some detail in section 2.3.1. They differ from Asset Expressions in the form in which the annotated information is provided, as annotation approaches usually use a different paradigm to provide the annotated information, e.g., [DM99, GS01, MYR03]. Specifically, this separation of the description paradigms for documents and annotations prohibits the recursive application of the annotation mechanism: Annotated information can in general not be annotated again. In this respect annotations are similar to the narrative worlds of MPEG-7 which exhibit the same property. Just like content in Asset Expressions, the format of the annotated documents can neither be assumed to include means to specify the base of an annotation nor can it be augmented with these means. In annotations this has the effect of a rather loose coupling between annotated document and annotation. Any annotation can be attached to any annotation base. Extensions, such as conceptual annotations [GS02], also carry out the creation of base and annotation in a single step, thus avoiding the question of correspondence of base and annotation by assuming that the author of the annotation has annotated correctly. Annotations are applicable to a wide variety of content kinds.

Central to the idea of markup is the embedding of information into the base document. This means that the document format must be prepared to accomodate the markup. The

information provided in the markup therefore is closely related to the document. In many cases the markup provides structural information on the document that facilitates the interpretation of the document. Some forms of markup are directed towards human users, others towards computers (see section 2.3.1 for examples). Markup for computers is often found in processing instructions, e.g., to specify how a document should appear in print or on screen. As the markup is embedded into the document, this integration is usually fully specified, for example in a grammar. This grammar also fixes the available markup means, thereby limiting its expressiveness. Markup can only be applied to a limited set of content formats. These formats need to provide means to markup the content. Thus markup can generally not be applied in retrospect to any content.

Hypermedia systems aim to interrelate a large body of content instances—called documents—by providing links between them. These links are usually not based on a whole source document but some part of it. In a special kind of hypermedia system—hypertext systems—this is especially prominent as the base of the link usually consists of only a few words from a large document. Hypermedia systems often store documents and links in an integrated format, similar to the embedding of markup in documents. However, there are exceptions to this rule in which links are first class citizens and are stored separately from the document they connect. An example of this are *Open Hypermedia Systems* [ØW96]. Other approaches add link types to differentiate between kinds of links [HYG99] or node types [BVA+97] to assign to documents. If the node types are enforced on the target documents of links, they are comparable to the semantic typing of abstraction variables in Asset Expressions.

Hypermedia systems are made for human users to improve the accessibility of documents by associative thought structures—an idea first put forward in 1945 [Bus45]—as compared to other means commonly available in information systems. If a link in a hypermedia system is used to explain the semantics of a portion of the base document, all the means are available to create the explaining document that could be used for the base document. However, the integration of the two documents is rather loose in most hypermedia systems. All the information that is available is that there exists a link. Systems thus have no means to detect and prevent senseless links between documents. Conceptual hypermedia systems attempt to remedy this situation by making both links and documents instances of a conceptual schema [NN91]. Links are then superimposed onto documents and the address of their base is stored explicitly, similarly to selectors in Asset Expressions. In principle, the hypermedia approach is sufficiently general to be applicable to any kind of media. In practice, particular implementations limit the available content kinds. Reasons can be specific needs of application domains, difficulties of embedding links in content (unless links are first class citizens), or implementation limitations.

The semantic web is an effort to facilitate information exchange by associating each resource on the world-wide web with a machine-processable semantic description [BLHL01]. Similarly to the narrative worlds of MPEG-7 this description contains a structured account of the content of the resource. However, unlike MPEG-7 or Asset Expressions, the resource is only described as a whole, no content components are available. The target audience are computers, humans are intended to use the semantic web through secondary interfaces for posing queries, etc.

The semantic web provides specialized means [ROH05, BHH+02] for descriptions. Most resources on the web, however, are not written in these description languages, resulting in a division between base resources (e.g., HTML pages or images) and semantic resources describing these. This redundancy of information is a commonly criticized aspect of the semantic web [HP02]. In principle, descriptions could be applied again to description documents such that the semantic resources are also described, but the focus of the semantic web is clearly on describe ordinary web resources and not on describing itself. Because resources are described as a whole, any resource that can be identified can also be described regardless of its content format. Relations between the semantic web and Conceptual Hypermedia Systems have also been explored [GBC+01].

Structured schemata define intensionally how entities can be described. To this end, classes or similar concepts are used. Individual descriptions are created as instances of these classes with the specified structure. Schemata are used in many circumstances, conceptual schemata are of particular interest here. They are mainly intended for use by computers, for example in the creation of information systems. In most cases there is also an indirect use of the schema by humans, for example through an information system. However, unlike the other approaches presented here, conceptual schemata usually do not explicitly deal with medial content. An exception are Asset-based schemata, which provide a dualistic model of entities with medial and conceptual descriptions. These schemata and their relationship with Asset Expressions have been discussed in the previous chapters (specifically 2.1.4, 6, and 7.2). Other schemata simply treat content as a piece of data with no particular semantics or capabilities. Therefore they also do not provide formalisms for integration of conceptual and medial data. However, means to interrelate entity descriptions exist, for example relationships in ER models [Che76].

9.2.2 Schema Creation Processes

Creating a conceptual schema for the application domain is a common task in software engineering. Various approaches to the task exist and are documented in literature. A central issue in most approaches is the question of how to obtain information about the domain from a domain expert. Many times, this information will then be codified by a modeling expert, who collaborates with the domain expert. The degree of involvement and collaboration of both domain and modeling expert varies, some approaches—such as the schema creation process UCSCP presented in section 7.2—can be enacted without a modeling expert. This section relates the UCSCP to processes with similar goals.

Object-oriented analysis is a common approach to software engineering [Som00]. Even though not all object-oriented modeling approaches contain an explicit conceptual model, some examples exist [Lar05, Fow04]. A differentiation is usually made between analysis and design models. The former describes what the system does, the latter describes how this functionality is achieved. Domain experts are usually involved in building the analysis model, but the understanding of the modeling expert is of great importance for the model. This is rather different to the UCSCP, which puts the focus on the domain expert by at least decoupling and in many cases removing the modeling expert altogether. Object-oriented approaches do not include particular means of feedback for the domain expert. It is assumed that this feedback can be communicated sufficiently by the modeling expert, who works with the domain expert to build the model. However, while some argue that the common representation paradigms for object-oriented models, such as UML class diagrams [BRJ99], can intuitively be understood by domain experts, such assumptions can be treacherous and lead to flaws in the model, see section 7.2.1.

Central issues of conceptual modeling are also found when creating an ontology for an application domain. Several processes for the development of ontologies have been proposed in literature [GPFLC04]. Two important problems of ontology engineering also occur in the conceptual modeling, in particular in the scenario of the UCSCP: The complicated reuse of existing ontologies and the complexity of modeling languages which can be impedimental to the involvement of domain experts. In the UCSCP the former problem occurs when the process starts from an existing model, e.g., during schema evolution. The latter problem is also found in conceptual modeling in general, as conceptual modeling paradigms are not necessarily more understandable to domain experts than their counterparts from ontologies.

Interesting for its similarity to UCSCP is the collaborative approach to ontological modeling of Holsapple and Joshi [HJ02] (abbreviated OMHJ below). It uses the Delphi method [LF75], which encourages collaborative decision making through a moderator as well as organized feedback. The process is divided in four phases: preparation (determining criteria and context of the ontology), anchoring (building an initial ontology as a starting point of iterative refinement), iterative improvement (by systematic collection of participants' opinions), and application (for

example in an information system). The UCSCP also uses iterative refinement of the model by posing questions about the model and using the answers to improve the model (phase II). The basis for these questions is—just like in OMHJ—an initial model. However, this model is build automatically in the case of UCSCP. The result is applied in a final fourth phase in both approaches. Collaboration of several users is a key element of OMHJ. It can be achieved in UCSCP as well, but this is not the prime focus. The UCSCP also differs from OMHJ in that is uses sample instances as the basis of modeling and also as a means for communication with domain experts. The paradigm to be used for communication with domain experts is not further elaborated by Holsapple and Joshi.

The On-To-Knowledge meta-process (section 2.2.3 or [SSSS01]) incorporates the development of an ontology into application development. Its result is an ontology for the domain of the application at hand. The process contains two iterative elements: refinement during the development of the ontology and evolution of the ontology, which is modeled as a part of application maintenance. During the refinement the domain expert gives input on the existing concepts in the ontology. This input is formalized by a modeling expert, who is familiar with the paradigm the ontology is expressed with. In the maintenance phase of the process, e.g., during the runtime of a system based on the created ontology, the need can arise to modify or augment the ontology. This is done by controlled refinement in a re-execution of the refinement phase. Both types of iteration can also be found in the UCSCP. The conceptual model is created iteratively based on user feedback in phase III. The process can return to the beginning from phase IV if the need for personalization or schema evolution arises. A central difference is that most activities of the On-To-Knowledge meta process can only be carried out by modeling experts.

Linguistic analysis is a methodology to obtain a conceptualization of a domain in textual documentation [Abb83]. Texts are analyzed for keywords, which form the basis of domain classes. In a second step attributes for and relationships between the classes are determined by similar means. In object-oriented analysis, the basis for linguistic analysis can be found in descriptions of use cases [Lar05, section 9.5]. The linguistic approach can also be applied to scenarios describing behavior [FKM04]. It is assumed that the basis for textual analysis is sufficiently verbose about the application domain to create a coherent domain model. If this is not the case, additional completion steps have to be inserted into the process [FKM04] to add the missing parts. In this expectation of completeness, linguistic analysis is similar to the UCSCP. However, the UCSCP takes as input expressions, which are more formal than natural language texts. This facilitates the extraction of relevant domain concepts, as it reduces the amount of information noise. Filtering such noise is a major problem in text-based approaches especially if the texts are written in a non-technical manner [AHKV98].

In general it can be observed that processes with a more formal approach to conceptual schema creation also expect more formality from the domain experts. Linguistic analysis requires relatively little formality in the input documents, which are given by domain experts. The extraction of schema classes from these texts is inspirational and requires an intuitive understanding of the application domain by the modeling expert. The UCSCP, which is based on Asset Expressions, takes as input expressions which are more formal than text. In return it can create conceptual schemata largely without input from a modeling expert.

9.3 Future Work

The modeling paradigms discussed in the comparison in the previous section can be divided into two groups: those made for human users and those made for computers. Key differences of the two groups are the degree of formality of the descriptions and the amount of information given in the model instances. Figure 9.2 shows a graph of this situation by plotting the formality of the model over the richness of its instances. Human users generally prefer models that only state what the user does not know yet—*incremental* models with respect to the user's contextual

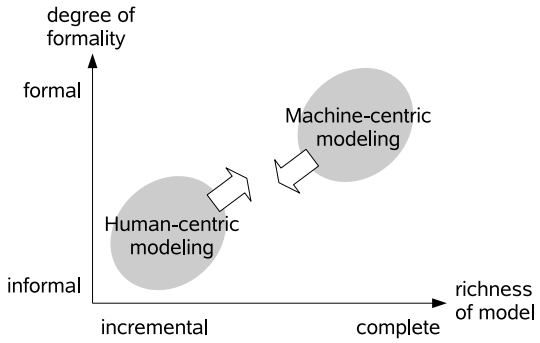


Figure 9.2: *Modeling paradigms differ in the degree of formality as well as in the amount of information conveyed in instances of the model (*richness*). Machine interpretation requires very rich and very formal models, human users are often more content with less formal models that only state what they do not know, i.e., do not contain a complete model of the application domain.*

knowledge—and a lesser degree of formality. Machine-centric models are characterized by a higher degree of formality, as well as more complete models, as contextual knowledge of the machine is usually assumed to be very small. Asset Expressions, linguistic analysis, and annotation techniques can be found towards the lower left corner in figure 9.2, descriptions using strict ontological means such as OWL [BHH+02], RDF [Bec04], and other technologies in the context of the semantic web towards the upper right. A common criticism of this division into human-centric and machine-centric modeling approaches is the necessity to provide duplicate descriptions using several description means in many cases. In fact, if one considers that not all members of the target audience—irrespective of whether they are computers or humans—share the same contextual knowledge, the need for even more descriptions of multiple levels of richness can arise. Asset Expressions take first steps towards providing multi-richness models by allowing the re-use of existing (less rich) models in enhanced descriptions. However, this is a one-way approach as there is no convenient way of weeding-out overly rich descriptions. Integration with computer-centric descriptions is also not straight-forward.

To reduce the need for redundant descriptions in order to suit various target audiences, an integrated model from which various descriptions can be created would be of great interest. This integrated model needs to support the creation of views on the descriptions. These views contain the right amount of information for their target audience and are expressed with an appropriate degree of formality. Based on their preferred view, users can then choose a matching interface paradigm (e.g., visual notation for humans or an object-representation for computers) to work with. Besides creating an integrated model, a major difficulty is the definition of views on descriptions. As the underlying integrated description needs to provide very complete information in order to satisfy the needs of even the most specific view, the view definition has to specify which parts to drop. Dropping too little makes it tedious for the user to discover the interesting parts of the description as the overlap with the user's contextual knowledge is very high. Dropping too much renders the description useless as there remain no points of contact which allow the user to integrate the description with present contextual knowledge. The latter is especially true for machine users. An approach to a solution could be the classification of the parts of descriptions according to Panofsky's levels of description (table 1.2 and [Pan70]) into pre-iconographical, iconographical, and iconological descriptions. Human users could then, for example, state in their views to remove pre-iconographical information as well as iconographical parts from domains they are familiar with. If the user is an expert in a particular domain, it might also be appropriate to drop iconological parts from this domain.

If views can be used to express the underlying descriptions in several paradigms, efforts will be needed to overcome the mismatches between the paradigms. Different types of conceptualization can cause great mismatch: Asset Expressions use semantic types, which have no structural consequences, but many ontological approaches as well as conceptual models use intensional classes for conceptualization. Converting between these is certainly non-trivial and possibilities need to be investigated carefully, see chapter 6. Another common mismatch is the assumption made about the completeness of the model. *Open world* models assume that the model is incomplete such that one cannot, for example, draw conclusions on the existance of real-world entities from the absence of corresponding instances in the model. On the contrary, *closed world* models assume that all relevant real-world entities are also present in the model. Some research is being done on the alignment of the two views by restrictions on the modeling formalism. Open-world description logics (DL) and closed-world Horn rules can be combined by requiring that every rule variable occurs in a non-DL-atom in the body of the rule [MSS05].

Ontological modeling means overlap with the functionality of Asset Expressions. Quite some knowledge is already codified in ontologies. Therefore making ontologies interoperable with or at least accessible from Asset Expressions is an interesting topic. Ontological modeling methodologies do not focus on medial descriptions but rather on structural ones as do for example the narrative worlds of MPEG-7. These are also a good example of existing integration with medial content.

Automation in Asset Expression Systems (AESs) can also be extended beyond the suggestion of semantic types by integrating existing work. Content components can be discovered automatically [HPS99, MYR03, PVS+06] providing the user with starting points for abstractions. This matches well with the notion of function signatures which abstract variables that are readily available in the body of the function. When creating a signature for a function, one does not have to first introduce the variables, but these are already present. In manually created Asset Expressions, however, the creator of the expression first has to provide the variables in the form of content components before abstractions can be made on them. Automatic segmentation of content into components provides the creators of expressions with variables to chose from. Naturally, not all variables will appear in the signature. For limited domains the segmantation can even classify the component—in Asset Expression terms by suggesting a semantic type for it [BDF+03].

Users can collaborate in AESs by referencing each other's expressions. This is currently only possible in a single system. It can, however, be expected that some expressions that are created during one project for a certain application domain can be reused in other projects dealing with related application domains. It is unlikely that all these projects can be carried out within the same system. Therefore, transferral of expressions to new projects and thereby to other systems would be beneficial. This raises questions of how the logistics of such transfers can be handled, i.e., how they affect the lifecycle of expressions in the original and remote systems. Collaborative distributed processes can be used to add value to content [SS99]. An expression is created in a certain context with a particular audience in mind. After it is transferred to another context its audience might be different, making it necessary to extend the contextual knowledge of the new audience to ensure understandability of the expression. This could, for example, be achieved by also transferring related expressions that contain the necessary additional information along with the expression. If carried out automatically the problems created here by gaps in contextual information are similar to those when defining views as described above. These issues in the logistics of entities can be attacked via *entity-closures* which describe the relevant context of an entity [Hup07]. This closure needs to be carried along to the target system.

Most conceptual schemata can express constraints on their instances. In fact, typing can also be interpreted as a special type of constraint [AWL94]. In conceptual schemata constraints are usually given on class level. They define conditions an instance has to meet in order to be a member of the class. Such definitions on class level allow convenient handling of a large

number of instances. However, in other areas constraints on instance level are also used, for example in clustering [WC00] or to allow simplified constraint definition by users [Bor86]. As an Asset Expression in general is not a member of a class, constraints need to be defined on expression level. This can either happen truely individually or constraints can be defined in traits to influence expressions created by trait. Interesting types of constraints include the restriction of application operands, the specification of a set of abstractions (as available in traits now), or the demand of particular content components. Because Asset Expressions do not have any built-in domains of literal values (e.g., natural numbers or character strings), constraints presently cannot be formulae that are based on computations in these domains. An interesting option would be to permanently associate constraints to an expression name. While the expression referenced by that name can still be redefined, all new bindings also need to satisfy the constraints. This can ensure the permanent applicability of expressions by this name in the context of all expressions the name is referenced from.

The UCSCP aids domain experts in creating a conceptual schema for a domain they have modeled in Asset Expressions. The expressions used in the process might have been created by multiple users, but the process itself needs to be run by a single user. This user then has to answer questions from the system, has to be trusted to detect any errors introduced during automatic schema construction, and has to be able to make all additional modifications either to the schema directly or to the expressions that favorably influence the final schema. These assumptions mean that a single user—or a group of users in one physical location—have to understand all aspects of the application domain well enough to take the required decisions. If a distributed team of users would like to collaboratively create a schema, some additional support from the system is required. First, the system needs to aggregate the expressions of all users, which is possible in current AESs. Second, the schema creation process itself must be implemented in a physically distributed manner. The current three-layer architecture of AESs is not sufficient for this as additional requirements arise in collaborative work on schemata such as concurrent modification issues and updates of distributed data [RR98]. Furthermore, communication among users needs to be encouraged. Such communication is much more difficult in a distributed setup than in a face-to-face scenario due to limited communication channels. Tools for collaborative distributed work on schemata have been proposed [RR98, CPP06]. Their incorporation into the interface for execution of the UCSCP can allow physically distributed schema creation.

The Asset Expression approach can be applied recursively to itself by using not just any content but providing an Asset Expression in the content. Abstractions over this content then talk about parts of the “content-expression”. This can be used to annotate expressions with various information, for example to facilitate collaboration. A selector language to address parts of Asset Expressions that are used as content is already in place: the Asset Expression Query Language. It can address any part of an expression, making it a good choice to implement selectors. An application example for higher-order Asset Expressions are user manuals that explain how to work with the Asset Expression approach or scenarios that capture expression provenance explicitly. Higher-order expressions can also be used to express traits by means to the cut and use component handlings.

Bibliography

[Abb83] Russell J. Abbott. Program design by Informal English Descriptions. Communications of the ACM, volume 26, no. 11; pages 882–894. 1983.

[ABCWM99] Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth Métais (editors). *Conceptual Modeling - ER '99, 18th International Conference on Conceptual Modeling, Paris, France, November, 15-18, 1999, Proceedings*, volume 1728 of *Lecture Notes in Computer Science*. Springer. 1999.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag. 1996.

[Ado04] Adobe Systems Incorporated. PDF Reference. Internet http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf. Accessed 10 Jan 2007. 2004.

[AFO05] Maristella Agosti, Nicola Ferro, and Nicola Orio. Annotating Illuminated Manuscripts: An Effective Tool for Research and Education. In *JCDL '05: Proceedings of the 5th ACM/IEEE-CS joint conference on Digital libraries*, pages 121–130. ACM Press, New York, NY, USA. 2005.

[AHKV98] Helena Ahonen, Oskari Heinonen, Mika Klemettinen, and A. Inkeri Verkamo. Applying Data Mining Techniques for Descriptive Phrase Extraction in Digital Document Collections. In *Proceedings of the Advances in Digital Libraries Conference, ADL '98*, pages 2–11. 1998.

[AK03] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. IEEE Software, volume 20, no. 5; pages 36–41. 2003.

[AL04] Marcelo Arenas and Leonid Libkin. A Normal Form for XML Documents. ACM Transactions on Database Systems, volume 29, no. 1; pages 195–232. 2004.

[All87] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge computer science texts. Cambridge University Press. 1987.

[Amb03] Scott W. Ambler. *Agile Database Techniques*. Wiley. 2003.

[ART90] Malcolm P. Atkinson, Philippe Richard, and Philip W. Trinder. Bulk Types for Large Scale Programming. In *East/West Database Workshop*, pages 228–250. 1990.

[AS83] Dana Angluin and Carl H. Smith. Inductive Inference: Theory and Methods. ACM Computing Surveys, volume 15, no. 3; pages 237–269. 1983.

[ASE02] Technical Standardization Committee on AV, IT Storage Systems, and Equipment. Exchangeable Image File Format for Digital Still Cameras: EXIF Version 2.2. http://www.exif.org/Exif2-2.PDF. 2002.

[AW85] Maryam Alavi and Ira R. Weiss. Managing the Risks Associated with End-User Computing. Jornal of Management Information Systems, volume 2, no. 3; pages 5–20. 1985.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft Typing with Conditional Types. In *Proceedings of the Conference on Principles of Programming Languages*, pages 163–173. 1994.

[Bar85] Hendrik Pieter Barendregt. *The Lambda Calculus*. Studies in Logic and the Foundations of Mathematics. North-Holland. 1985.

[BB04] Alexander Borgida and Ronald J. Brachman. *Description Logics Handbook*, chapter Conceptual Modeling with Description Logics, pages 359–381. Cambridge University Press. 2004.

[BBF+02] Marcia J. Bates, Harry Bruce, Raya Fidel, Peter Ingwersen, and Pertti Vakkari. Speculations on browsing, directed searching, and linking in relation to the Bradford distribution. In *CoLIS 4 : fourth international conference on conceptions of library and information science : emerging frameworks and methods*. Seattle WA. 2002.

[BCF+05] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. Internet http://www.w3.org/TR/xquery/. Accessed 30 December 2006. 2005.

[BCM99] Paolo Bottoni, Maria Francesca Costabile, and Pierro Mussio. Specification and dialogue control of visual interaction through visual rewriting systems. ACM Transactions Programming Languages and Systems, volume 21, no. 6; pages 1077–1136. 1999.

[BDF+03] Kobus Barnard, Pinar Duygulu, David Forsyth, Nando de Freitas, David M. Blei, and Michael I. Jordan. Matching words and pictures. J. Mach. Learn. Res., volume 3; pages 1107–1135. 2003.

[Bec04] Dave Beckett. RDF/XML Syntax Specification (Revised). Internet http://www.w3.org/TR/rdf-syntax-grammar/. Accessed 18 October 2006. 2004.

[Ber03] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *First Conference on Innovative Data Systems Research 2003*. 2003.

[BHH+02] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. MucGuinness Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language 1.0. Internet http://www.w3.org/TR/owl-ref/. Accessed 16 Dec 2006. 2002.

[BHLT06] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.0 (Second Edition). http://www.w3.org/TR/REC-xml-names/. 2006.

[BJJW97] Magnus Boman, Janis A. Bubenko Jr., Paul Johannesson, and Benkt Wangler. *Conceptual Modelling*. Prentice Hall. 1997.

[BLCGP92] Tim Berners-Lee, Robert Cailliau, Jean-Francois Groff, and Bernd Pollermann. World-Wide Web: The Information Universe. Electronic Networking: Research, Applications and Policy, volume 1, no. 2; pages 74–82. 1992.

[BLFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifier (URI): Generic Syntax. Request for Comments 3986. 2005.

[BLHL01] Tim Berners-Lee, James Handler, and Ora Lassila. The Semantic Web. Scientific American, pages 34–43. 2001.

[BMNPS04] Franz Bader, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider (editors). *Description Logics Handbook*. Cambridge University Press. 2004.

[BMS84] Michael L. Brodie, John Mylopoulos, and Joachim W. Schmidt (editors). *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Topics in Information Systems. Springer-Verlag. 1984.

[BN04] Paul De Bra and Wolfgang Nejdl (editors). *Adaptive Hypermedia and Adaptive Web-Based Systems, Third International Conference, AH 2004, Proceedings*, volume 3137 of *Lecture Notes in Computer Science*. Springer. 2004.

[Boe07] Boemie Project. Project website. Internet http://www.boemie.org/. Accessed 16 Jan 07. 2007.

[Bor85] Alexander Borgida. Features of Languages for the Development of Information Systems at the Conceptual Level. ICCC Software, volume 2, no. 1. 1985.

[Bor86] Alan Borning. Defining constraints graphically. ACM SIGCHI Bulletin, volume 17, no. 4; pages 137–143. 1986.

[Bos04] Sebastian Bossung. *Generating Schema Information for Views over Semistructured Data*. Master's thesis, Technische Universität Hamburg-Harburg. 2004.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Fran cois Yergeau, and John Cowan. XML 1.1 (Second Edition). Internet http://www.w3.org/TR/2006/REC-xml11-20060816/. Accessed 18 October 2006. 2006.

[Bra95] Rebecca W. Bray. USML-2 Astronauts. Internet http://liftoff.msfc.nasa.gov/Shuttle/USML2/crew/crew.html. Accessed 10 Jan 2007. 1995.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley. 1999.

[Bru72] Nicolaas de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. Indagationes Mathematicae, volume 5, no. 34; pages 381–392. 1972.

[BS82] Michael L. Brodie and Joachim W. Schmidt. Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group. SIGMOD Record, volume 12, no. 4; pages i–62. 1982.

[BSG+04] Sebastian Bossung, Hermann Stoeckle, John C. Grundy, Robert Amor, and John G. Hosking. Automated Data Mapping Specification via Schema Heuristics and User Interaction. In *Proceedings of the Automated Software Engineering Conference, 2004*, pages 208–217. IEEE Computer Society. 2004.

[BSHS06] Sebastian Bossung, Hans-Werner Sehring, Patrick Hupe, and Joachim W. Schmidt. Open and Dynamic Schema Evolution in Content-Intensive Web Applications. In Cordeiro et al. [CPEF06], pages 109–116. 2006.

[BSS05] Sebastian Bossung, Hans-Werner Sehring, and Joachim W. Schmidt. Conceptual Content Management for Enterprise Web Services. In J. Akoka, S.W. Liddle, I.-Y. Song, M. Bertolotto, I. Comyn-Wattiau, W.-J. van den Heuvel, M. Kolp, J. Trujillo, C. Kop, and H.C. Mayr (editors), *Perspectives in Conceptual Modeling*, volume 3770 of *Lecture Notes in Computer Science*, pages 343–353. Springer-Verlag. 2005.

[BSSS05] Sebastian Bossung, Hans-Werner Sehring, Michael Skusa, and Joachim W. Schmidt. Conceptual Content Management for Software Engineering Processes. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam (editors), *Advances in Databases and Information Systems, 2005*, volume 3631 of *Lecture Notes in Computer Science*, pages 309–324. Springer-Verlag. 2005.

[Bun97] Peter Buneman. Semistructured Data. In *Principles of Database Systems*, pages 117–121. ACM Press. 1997.

[Bus45] Vannevar Bush. As We May Think. The Atlantic Monthly, volume 176, no. 1; pages 101–108. 1945.

[BVA+97] Michael Bieber, Fabio Vitali, Helen Ashman, Venkatraman Balasubramanian, and Harri Oinas-Kukkonen. Fourth generation hypermedia: some missing links for the World Wide Web. International Journal of Human-Computer Studies, volume 47, no. 1; pages 31–65. 1997.

[BVNK01] Peter Bosch, Arjen de Vries, Niels Nes, and Martin Kersten. A Case for Image Quering through Image Spots. In *Storage and Retrieval for Media Databases 2001*, volume 4315 of *Proceedings of SPIE*, pages 20–30. San Jose, USA. 2001.

[CACW02] Samira Si-Said Cherfi, Jacky Akoka, and Isabelle Comyn-Wattiau. Conceptual Modeling Quality - From EER to UML Schemas Evaluation. In Spaccapietra et al. [SMK02], pages 414–428. 2002.

[Cas00] Ernst Cassirer. *The Logic of the Cultural Scienes*. Yale University Press, New Haven. 2000.

[Cas01] Ernst Cassirer. *Die Sprache*, volume 11 of *Philosophie der symbolischen Formen*. Felix Meiner Verlag GmbH. 2001.

[Cas05] David Plans Casal. Advanced Software Development for Web Applications. Technical Report TSW0505, JISC Technology and Standards Watch. 2005.

[Cat94] R.G.G. Cattel (editor). *The Object Database Standard ODMG-93*. Morgan Kaufman. 1994.

[CCC+04] Andrea Calì, Diego Calvanese, Simona Colucci, Tommaso Di Noia, and Francesco M. Donini. A Description Logic Based Approach for Matching User Profiles. In Volker Haarslev and Ralf Möller (editors), *Description Logic Workshop, 2004*, pages 110–119. 2004.

[CCFS95] M. Sheelagh T. Carpendale, David J. Cowperthwaite, F. David Fracchia, and Thomas C. Shermer. Graph Folding: Extending Detail and Context Viewing into a Tool for Subgraph Comparisons. In Franz J. Brandenburg (editor), *Proc. 3rd Int. Symp. Graph Drawing, GD*, 1027, pages 127–139. Springer-Verlag, Berlin, Germany. 1995.

[Che76] Peter P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. ACM Transactions on Database Systems, volume 1, no. 1; pages 9–36. 1976.

[CKM02] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: the Tropos project. Information Systems, volume 27, no. 6; pages 365–389. 2002.

[CMF96] Yves Chiaramella, Philippe Mulhem, and Franck Fourel. A model of multimedia information retrieval. Technical Report FERMI ESPRIT BRA 8134, University of Glasgow. 1996.

[CNS+04] Simona Colucci, Tommaso Di Noia, Eugenio Di Sciascio, Marina Mongiello, and Francesco M. Donini. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an e-marketplace. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 41–50. ACM Press, New York, NY, USA. 2004.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM, volume 13, no. 6; pages 377–387. 1970.

[Con87] Jeff Conklin. Hypertext: An Introduction and Survey. IEEE Computer, volume 20, no. 9; pages 17–41. 1987.

[CPEF06] José A. Moinhos Cordeiro, Vitor Pedrosa, Bruno Encarnação, and Joaquim Filipe (editors). *WEBIST 2006, Proceedings of the Second International Conference on Web Information Systems and Technologies: Internet Technology / Web Interface and Applications, Setúbal, Portugal, April 11-13, 2006*. INSTICC Press. 2006.

[CPP06] Weiqin Chen, Roger Pedersen, and Øystein Pettersen. Building UML Models Collaboratively. In José A. Moinhos Cordeiro, Vitor Pedrosa, Bruno Encarnação, and Joaquim Filipe (editors), *WEBIST 2006, Proceedings of the Second International Conference on Web Information Systems and Technologies (2)*, pages 106–111. INSTICC Press. 2006.

[CR94] Vassilis Christophides and Antoine Rizk. Querying Structured Documents with Hypertext Links using OODBMS. In *European Conference on Hypertext*, pages 186–197. 1994.

[CR03] Youngok Choi and Edie M. Rasmussen. Searching for images: The analysis of users' queries for image retrieval in American history. JASIST, volume 54, no. 6; pages 498–511. 2003.

[CRD87] James H. Coombs, Allen H. Renear, and Steven J. DeRose. Markup systems and the future of scholarly text processing. Communications of the ACM, volume 30, no. 11; pages 933–947. 1987.

[DBO06] Patrick Durusau, Michael Brauer, and Lars Oppermann. Open Document Format for Office Applications. OASIS draft. 2006.

[DKM+05] Lois M. L. Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor (editors). *Conceptual Modeling - ER 2005, 24th International Conference on Conceptual Modeling, Klagenfurt, Austria, October 24-28, 2005, Proceedings*, volume 3716 of *Lecture Notes in Computer Science*. Springer. 2005.

[DM99] Lois M. L. Delcambre and David Maier. Models for Superimposed Information. In *ER '99: Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 264–280. Springer-Verlag, London, UK. 1999.

[DMFP05] Isabel Díaz, Lidia Moreno, Inmaculada Fuentes, and Oscar Pastor. Integrating Natural Language Techniques in OO-Method. In Alexander F. Gelbukh (editor), *CICLing*, volume 3406 of *Lecture Notes in Computer Science*, pages 560–571. Springer. 2005.

[Dor03] Dov Dori. The Visual Semantic Web: Unifying Human and Machine Semantic Web Representations with Object-Process Methodology. In Isabel F. Cruz, Vipul Kashyap, Stefan Decker, and Rainer Eckstein (editors), *Semantic Web and Databases Workshop*, pages 415–433. 2003.

[DT89] János Demetrovics and Bernhard Thalheim (editors). *MFDBS 89, 2nd Symposium on Mathematical Fundamentals of Database Systems, Visegrád, Hungary, June 26-30, 1989*, volume 364 of *Lecture Notes in Computer Science*. Springer. 1989.

[DTOP02] Magali Dubosson-Torbay, Alexander Osterwalder, and Yves Pigneur. eBusiness Model Design, Classification and Measurements. Thunderbird International Business Review, volume 44, no. 1; pages 5–23. 2002.

[Ell04] James Elliott. *Hibernate: A Developer's Notebook*. OREILLY. 2004.

[EMK+04] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. SQL: 2003 has been published. SIGMOD Record, volume 33, no. 1; pages 119–126. 2004.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings. 1994.

[EP00] José Esteves and Joan Antoni Pastor. Enterprise Resource Planning Systems Research: An Annotated Bibliography. Communications of Association for Information Systems, volume 7, no. 8; pages 1–52. 2000.

[ES00] Michael Erdmann and Rudi Studer. How to Structure and Access XML Documents With Ontologies. Data and Knowledge Engineering. 2000.

[eXi] eXist – Open Source Native XML Database. Website http://www.exist-db.org/. Accessed 19 October 2006.

[Fel98] Christiane Fellbaum (editor). *WordNet: An Electronic Lexical Database*. The MIT Press. 1998.

[FFM04] Daniela Fogli, Giuseppe Fresta, and Piero Mussio. On Electronic Annotation and its Implementation. In Maria Francesca Costabile (editor), *Proceedings of the Working Conference on Advanced Visual Interfaces, AVI 2004*, pages 98–102. ACM Press. 2004.

[FKM04] Günther Fliedl, Christian Kop, and Heinrich C. Mayr. From Textual Scenarios to a Conceptual Schema. Data & Knowledge Engineering, volume 55, no. 1; pages 20–37. 2004.

[Fow04] Martin Fowler. *UML Distilled*. Addison-Wesley, 3 edition. 2004.

[FW04] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. Internet http://www.w3.org/TR/xmlschema-0/. Accessed on 15 Dec 2006. 2004.

[Gar65] Newton Garver. Varieties of Use and Mention. Philosophy and Phenomenological Research, volume 26, no. 2; pages 230–238. 1965.

[GB05] Margaret Graham and Christopher Bailey. Digital Images and Art Historians – Compare and Contrast Revisited. In *Proceedings of Digital Resources for the Humantities 2005*, pages 21–24. 2005.

[GBC+01] Carole A. Goble, Sean Bechhofer, Les Carr, David De Roure, and Wendy Hall. Conceptual Open Hypermedia = The Semantic Web? In Stefan Decker, Dieter Fensel, Amit Sheth, and Steffen Staab (editors), *The Second International Workshop on the Semantic Web*. 2001.

[GGR+00] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: A System for Extracting Document Type Descriptors from XML Documents. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (editors), *SIGMOD Conference*, pages 165–176. ACM. 2000.

[GH04] Joseph Goguen and Fox Harrell. Foundations for Active Multimedia Narrative: Semiotic spaces and structural blending. Internet at http://www.cs.ucsd.edu/users/goguen/pps/narr.pdf. Accessed 20 Mar 2007. 2004.

[GJP00] Marcela Genero, Luis Jiménez, and Mario Piattini. Measuring the Quality of Entity Relationship Diagrams. In Delcambre et al. [DKM+05], pages 513–526. 2000.

[GNA99] Mohamed M. Gammoudi, Ibtissem Nafkha, and Z. Abdelouahab. An Incremental and Semi-automatic Method Inheritance Graph Hierarchy Construction. In Akoka et al. [ABCWM99], pages 31–46. 1999.

[Gog04] Joseph A. Goguen. Ontology, Society, and Ontotheology. In Achille Varzi and Laure Vieu (editors), *Proceedings of the International Conference on Formal Ontologies in Informatin Systems*, pages 95–103. IOS Press. 2004.

[Gol90] Charles F. Goldfarb. *The SGML Handbook*. Oxford University Press. 1990.

[GPFLC04] Asunción Gómez-Pérez, Mariano Fernández-López, and Oscar Corcho. *Ontological Engineering*. Springer. 2004.

[Gru93] Thomas R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. International Journal of Human-Computer Studies, volume 43, no. 5-6; pages 907–928. 1993.

[GS01] Michael Gertz and Kai-Uwe Sattler. A Model and Architecture for Conceptualized Data Annotations. Technical Report CSE-2001-11, Department of Computer Science, University of California. 2001.

[GS02] Michael Gertz and Kai-Uwe Sattler. Integrating Scientific Data through External, Concept-based Annotations. In Zoé Lacroix (editor), *Second International Workshop Data Integration over the Web*, pages 87–102. University of Toronto Press. 2002.

[GSG+02] Michael Gertz, Kai-Uwe Sattler, Fred Gorin, Michael Hogarth, and Jim Stone. Annotating Scientific Images: A Concept-based Approach. In *14th International Conference on Scientific and Statistical DatabaseManagement*, pages 59–68. IEEE Computer Society. 2002.

[GST00] Dina Q. Goldin, Srinath Srinivasa, and Bernhard Thalheim. IS = DBS + Interaction: Towards Principles of Information System Design. In Delcambre et al. [DKM+05], pages 140–153. 2000.

[Gua98] Nicola Guarino. Formal Ontology in Information Systems. In N. Guarino (editor), *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems*, pages 3–15. Trento, Italy. 1998.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (editors), *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann. 1997.

[GW02] Nicola Guarino and Christopher A. Welty. Evaluating Ontological Decisions with OntoClean. Communications of the ACM, volume 45, no. 2; pages 61–65. 2002.

[GWG06] Andreas Gruber, Rupert Westenthaler, and Eva Gahleitner. Supporting Domain Experts in Creating Formal Knowledge Models (Ontologies). In Klaus Tochtermann und Hermann Maurer (editor), *Proceedings of I-KNOW'06. 6th International Conference on knowledge management (2006)*, pages 252–260. Graz, Austria. 2006.

[Hal01] Terry Halpin. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design.* Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann. 2001.

[HBHV04] Samira Hammiche, Salima Benbernou, Mohand-Said Hacid, and Athena Vakali. Semantic Retrieval of Multimedia Data. In *MMDB '04: Proceedings of the 2nd ACM international workshop on Multimedia databases*, pages 36–44. ACM Press, New York, USA. 2004.

[HBR93] Lynda Hardman, Dick C. A. Bulterman, and Guido van Rossum. Links in Hypermedia: The Requirement for Context. In *Hypertext*, pages 183–191. ACM. 1993.

[HG02] Farshad Hakimpour and Andreas Geppert. Global Schema Generation Using Formal Ontologies. In Spaccapietra et al. [SMK02], pages 307–321. 2002.

[HJ02] Clyde W. Holsapple and Kshiti D. Joshi. A Collaborative Approach to Ontology Design. Communications of the ACM, volume 45, no. 2; pages 42–47. 2002.

[HKM93] Gerd G. Hillebrand, Paris C. Kanellakis, and Harry G. Mairson. Database Query Languages Embedded in the Typed Lambda Calculus. In *IEEE Symposium on Logic in Computer Science*, pages 332–343. IEEE Computer Society. 1993.

[HP02] Stefan Haustein and Jörg Pleumann. Is Participation in the Semantic Web Too Difficult? In Ian Horrocks and James A. Hendler (editors), *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 448–453. Springer. 2002.

[HP06] Jerry R. Hobbs and Feng Pan. Time Ontology in OWL. Internet http://www.w3.org/TR/owl-time/. Accessed 1 Nov 2006. 2006.

[HPS99] Knut Hartmann, Bernhard Preim, and Thomas Strothotte. Describing Abstraction in Rendered Images through Figure Captions. Electronic Transactions on Artificial Intelligence, volume 3, no. A; pages 1–26. 1999.

[HTS+06] Alaa Halawani, Alexandra Teynor, Lokesh Setia, Gerd Brunner, and Hans Burkhardt. Fundamentals and Applications of Image Retrieval: An Overview. Datenbank-Spektrum, volume 18; pages 14–23. 2006.

[Hun01] Jane Hunter. Adding Multimedia to the Semantic Web: Building an MPEG-7 Ontology. In *Proceedings of the 1st International Semantic Web Working Symposium*, pages 261–283. 2001.

[Hup07] Patrick Hupe. *Prozesse über heterogenen Arbeitsumgebungen: Asset-basiertes Modell und Systemarchitektur*. Ph.D. thesis, Techische Universität Hamburg-Harburg. To appear. 2007.

[HYG99] Klaus Marius Hansen, Christian Yndigegn, and Kaj Grønbæk. Dynamic Use of Digital Library Material - Supporting Users with Typed Links in Open Hypermedia. In Serge Abiteboul and Anne-Marie Vercoustre (editors), *ECDL*, volume 1696 of *Lecture Notes in Computer Science*, pages 254–273. Springer. 1999.

[Int01] International Organisation of Standardisation. Information Technology – Multimedia Content Description Interface – Part 4: Audio. 15938-4:2001(E). 2001.

[Int04] International Organisation for Standardisation. MPEG-7 Overview. Internet http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm. Accessed 15 Dec 2006. 2004.

[Kat05] Susumu Katayama. Systematic Search for Lambda Expressions. In *Sixth Symposium on Trends in Functional Programming (TFP2005)*, pages 195–205. 2005.

[Kei94] Lucinda H. Keister. User Types and Queries: Impact on Image Access Systems. In Raya Fidel, Trudi Bellardo Hahn, Edie M. Rasmussen, and Philip Philip J. Smith (editors), *Challenges in Indexing Electronic Text and Images*, pages 7 – 22. 1994.

[Kes02] Stephan Kesper. A Proof of the Turing-completeness of XSLT and XQuery. Technical report, University of Tübingen. 2002.

[KM05] Hyoung-Gook Kim and Nicolas Moreau. *MPEG-7 Audio and Beyond*. John Wiley & Sons, Ltd. 2005.

[Kos02] Harald Kosch. MPEG-7 and Multimedia Database Systems. SIGMOD Record, volume 31, no. 2; pages 34–39. 2002.

[KS98] Vipul Kashyap and Amit Sheth. Semantic Heterogeneity in Global Information System: The Role of Metadata, Context and Ontologies. In M. Papazoglou and G. Schlageter (editors), *Cooperative Information Systems: Current Trends and Directions*. Academic Press, London. 1998.

[KS03] Yannis Kalfoglou and Marco Schorlemmer. Ontology Mapping: The State of the Art. Knowledge Engineering Review, volume 18, no. 1; pages 1–31. 2003.

[Kwa92] Barbara H. Kwasnik. A Descriptive Study of the Functional Components of Browsing. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 191–203. North-Holland. 1992.

[Lam86] Leslie Lamport. *Latex: A Document Preparation System*. Addison-Wesley. 1986.

[Lar05] Craig Larman. *Object-oriented Analsyis and Design and Iterative Development*. Prentice Hall, 3 edition. 2005.

[Lay88] Sara Shatford Layne. Analyzing the Subject of a Picture: A Theoretical Approach. Cataloguing and Classication, volume 6; pages 39–62. 1988.

[LBF+06] C. Lutz, F. Baader, E. Franconi, D. Lembo, R. Möller, R. Rosati, U. Sattler, B. Suntisrivaraporn, and S. Tessaris. Reasoning Support for Ontology Design. In B. Cuenca Grau, P. Hitzler, C. Shankey, and E. Wallace (editors), *In Proceedings of the second international workshop OWL: Experiences and Directions*. To appear. 2006.

[LBK03] Mathias Lux, Jutta Becker, and Harald Krottmaier. Caliph & Emir: Semantic Annotation and Retrieval in Personal Digital Photo Libraries. In *CAiSE 03 Forum at 15th Conference on Advanced Information Systems Engineering*, pages 85–89. Velden, Austria. 2003.

[LF75] Harold A. Lindstone and Murray Furoff. *The Delphi Method: Techniques and Applications*. Addison-Wesley. 1975.

[LF01] Simon Lok and Steven Feiner. A Survey of Automated Layout Techniques for Information Presentations. In *Proceedings of the International Symposium on Smart Graphics 2001*, pages 61–68. 2001.

[LÖSO97] John Z. Li, M. Tamer Özsu, Duane Szafron, and Vincent Oria. MOQL: A Multimedia Object Query Language. Technical Report TR-97-01, Department of Computing Science, University of Alberta. 1997.

[LS87] Peter C. Lockemann and Joachim W. Schmidt (editors). *Datenbankhandbuch.* Springer. 1987.

[Man91] Rainer Manthey. Towards a Unified View of Query- and Update-driven Inference in Deductive Databases. In *DAISD*, pages 220–224. 1991.

[Mar88] Karen Markey. Access to Iconographical Research Collections. Library Trends, volume 37, no. 2; pages 154–174. 1988.

[Mar02a] Jose M. Martinez. MPEG-7: Overview of MPEG-7 Description Tools, Part 2. IEEE Multimedia, pages 83–93. 2002.

[Mar02b] Jose M. Martinez. MPEG-7: The Generic Multimedia Description Standard, Part 1. IEEE Multimedia, pages 78–87. 2002.

[Mat93] Florian Matthes. *Persistente Objektsysteme*. Springer Verlag. 1993.

[May02] Heinrich C. Mayr. Do We Need an Ontology of Ontologies? In Spaccapietra et al. [SMK02], page 15. 2002.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 706–706. ACM Press, New York, NY, USA. 2006.

[McG03] Deborah L. McGuinness. Ontologies Come of Age. In Dieter Fensel, James A. Hendler, Henry Lieberman, and Wolfgang Wahlster (editors), *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, pages 171–194. MIT Press. 2003.

[McL94] Marshall McLuhan. *Understanding Media: The Extensions of Man*. The MIT Press, Cambridge. 1994.

[McL02] Brett McLaughlin. *Java & XML Data Binding*. O'Reilly. 2002.

[MCM05] Jean Martinet, Yves Chiaramella, and Philippe Mulhem. A Model for Weighting Image Objects in Home Photographs. In Otthein Herzog, Hans-Jörg Schek, Norbert Fuhr, Abdur Chowdhury, and Wilfried Teiken (editors), *Proceedings of the ACM Conference on Information and Knowledge Management*, pages 760–767. ACM. 2005.

[Mec95] Mourad Mechkour. A Multifacet Formal Image Model for Information Retrieval. In Ian Ruthven (editor), *Proceedings of the Workshop on Multimedia Information Retrieval*, Workshops in Computing. BCS. 1995.

[Mei02] Wolfgang Meier. eXist: An Open Source Native XML Database. In Akmal B. Chaudri, Mario Jeckle, Erhard Rahm, and Rainer Unland (editors), *Web, Web-Services, and Database Systems. NODe 2002 Web- and Database-Related Workshops*, volume 2593, pages 169–183. Springer LNCS Series. 2002.

[MGMW05] David E. Millard, Nicholas Gibbins, Danius T. Michaelides, and Mark J. Weal. Mind the Semantic Gap. In Siegfried Reich and Manolis Tzagarakis (editors), *Hypertext*, pages 54–62. ACM. 2005.

[Mic06] Microsoft, Inc. Office 2003 XML Reference Schemas. Internet. Http://www.microsoft.com/office/xml/overview.mspx. 2006.

[MJF03] Neil A.M. Maiden, Sara Jones, and Mary Flynn. Innovative Requirements Engineering Applied to ATM. In *Proceedings Air Traffic Management*. Budapest. 2003.

[MK98] Heinrich C. Mayr and Christian Kop. Conceptual Predesign – Bridging the Gap between Requirements and Conceptual Design. In *Proceedings of the Third International Conference on Requirements Engineering*, pages 90–100. IEEE Computer Society. 1998.

[MM03] Joaquin Miller and Jishnu Mukerji. *MDA Guide*. Object Management Group, Inc. 2003.

[MM04] Frank Manola and Eric Miller. RDF Primer. Internet http://www.w3.org/TR/rdf-primer/. Accessed 18 October 2006. 2004.

[MMWR01] Danius T. Michaelides, David E. Millard, Mark J. Weal, and David De Roure. Auld Leaky: A Contextual Open Hypermedia Link Server. In Siegfried Reich, Manolis Tzagarakis, and Paul De Bra (editors), *OHS-7/SC-3/AH-3*, volume 2266 of *Lecture Notes in Computer Science*, pages 59–70. Springer. 2001.

[MO04] Jonathan Marsh and David Orchard. XML Inclusions (XInclude) Version 1.0. Internet http://www.w3.org/TR/xinclude/. Accessed 16 Dec 2006. 2004.

[Moo05] Daniel L. Moody. Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions. Data & Knowledge Engineering, volume 55, no. 3; pages 243–276. 2005.

[Mot03] Motion Picture Experts Group. MPEG-7. ISO/IEC 15938-6:2003. 2003.

[MPP00] Annamaria Musto, Giuseppe Polese, and A. Pannella. Automatic Generation of RDBMS Based Applications from Object Oriented Design Schemes. In *SAC (1)*, pages 398–402. 2000.

[MS91] Florian Matthes and Joachim W. Schmidt. Bulk Types: Built-In or Add-On? In Paris C. Kanellakis and Joachim W. Schmidt (editors), *DBPL*, pages 33–54. Morgan Kaufmann. 1991.

[MS01] Alexander Maedche and Steffen Staab. Ontology Learning for the Semantic Web. IEEE Intelligent Systems, volume 16, no. 2; pages 72–79. 2001.

[MSS05] Boris Motik, Ulrike Sattler, and Rudi Studer. Query Answering for OWL-DL with rules. Journal of Web Semantics, volume 3, no. 1; pages 41–60. 2005.

[Mue07] Kai Mueller. *Signature Matching by Concept Contraction and Abduction*. Master's thesis, Techische Universität Hamburg-Harburg. To appear. 2007.

[Mus98] Mark M. Musen. Ontology-Oriented Design and Programming. In J. Cuena, Y. Demazeau, A. Garcia, and J. Treur (editors), *Knowledge Engineering and Agent Technology*. IOS Press. 1998.

[MW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. ACM Transaction on Softwware Engineering Methodology, volume 6, no. 4; pages 333–369. 1997.

[MYR03] Saikat Mukherjee, Guizhen Yang, and I.V. Ramakrishnan. Automatic Annotation of Content-Rich HTML Documents: Structural and Semantic Analysis. In *The SemanticWeb - ISWC 2003*, volume 2870 of *Lecture Notes in Computer Science*, pages 533–549. 2003.

[Nel65] Ted Nelson. Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate. In *Proceedings of the 1965 20th National Conference*, pages 84–100. ACM Press, New York, NY, USA. 1965.

[Ng00] Gary Kwok-Chu Ng. *Interactive Visualisation Techniques for Ontology Development*. Ph.D. thesis, University of Manchester, Department of Computer Science. 2000.

[NH97] Natalya Fridman Noy and Carole D. Hafner. The State of the Art in Ontology Design: A Survey and Comparative Review. AI Magazine, volume 18, no. 3; pages 53–74. 1997.

[NJ02] Andrew Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *Proceedings of 5th International Workshop on the Web and Databases*, pages 61–66. Madison, WI. 2002.

[NM06] Bernd Neumann and Ralf Möller. On Scene Interpretation with Description Logics. In H.I. Christensen and H.-H. Nagel (editors), *Cognitive Vision Systems: Samping the Spectrum of Approaches*, number 3948 in LNCS, pages 247–278. Springer. 2006.

[NN91] Jocelyne Nanard and Marc Nanard. Using Structured Types to Incorporate Knowledge in Hypertext. In *Hypertext'91 Proceedings, San Antonio, Texas, USA*, pages 329–343. ACM. 1991.

[Nor95] Moira C. Norrie. Distinguishing Typing and Classification in Object Data Models. In *Information Modelling and Knowledge Bases*, volume VI. IOS. 1995.

[NSS01] Katashi Nagao, Yoshinari Shirai, and Kevin Squire. Semantic Annotation and Transcoding: Making Web Content More Accessible. IEEE Multimedia, volume 8, no. 2; pages 69–81. 2001.

[OAM99] Ilia A. Ovsiannikov, Michael A. Arbib, and Thomas H. McNeill. Annotation technology. International Journal of Human-Computer Studies, volume 50, no. 4; pages 329–362. 1999.

[OGC04] Open Geospacial Consortium OGC. Geography Markup Language (GML). Internet http://portal.opengeospatial.org/files/?artifact_id=4700. Accessed 1 Nov 2006. 2004.

[OMG96] Object Management Group OMG. The Common Object Request Broker: Architecture and Specification Revision 2.0. OMG TC Document 96.03.04. 1996.

[Org86] International Standards Organization. Information Processing. Text and Office Systems. Standard Generalized markup Language (SGML): Draf Standard. ISO 8879:1986. 1986.

[ØW96] Kasper Østerbye and Uffe Kock Wiil. The Flag Taxonomy of Open Hypermedia Systems. In *Hypertext*, pages 129–139. ACM. 1996.

[Pan70] Erwin Panofsky. *Meaning in the Visual Arts*. Penguin. 1970.

[PBE98] Simon Pollitt, Andrew Burrow, and Peter W. Eklund. WebKB-GE - A Visual Editor for Canonical Conceptual Graphs. In *International Conference on Conceptual Structures*, pages 111–118. 1998.

[Pei31] Charles Sanders Peirce. *Collected Papers of Charles Sanders Peirce*. Harvard University Press. 1931.

[PGC02] Domenico M. Pisanelli, Aldo Gangemi, and Gerardo Steve Consiglio. Ontologies and Information Systems: the Marriage of the Century? In *Proceedings of the Lyee Workshop*. Paris. 2002.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press. 2002.

[PMM05] Angel Puerta, Michael Micheletti, and Alan Mak. The UI Pilot: A Model-based Tool to Guide Early Interface Design. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 215–222. ACM Press, New York. 2005.

[Pri96] Lynne A. Price. Practical SGML as an Introduction to SGML. SIGDOC Asterisk Journal of Computer Documentation, volume 20, no. 2; pages 36–38. 1996.

[Pro97] H. A. Erik Proper. Data Schema Design as a Schema Evolution Process. Data Knowledge Engineering, volume 22, no. 2; pages 159–189. 1997.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 146–161. 1991.

[PSZ99] Christine Parent, Stefano Spaccapietra, and Esteban Zimányi. Spatio-Temporal Conceptual Models: Data Structures + Space + Time. In Claudia Bauzer Medeiros (editor), *ACM-GIS*, pages 26–33. ACM. 1999.

[PVS+06] Ioannis Pratikakis, Iris Vanhamel, Hichem Sahli, Basilios Gatos, and Stavros J. Perantonis. Unsupervised Watershed-driven Region-based Image Retrieval. IEEE Proceedings on Vision, Image and Signal Processing, Special Issue on Knowledge-based Digital Media Processing, volume 153, no. 3; pages 313–322. 2006.

[PW97] Thomas A. Phelps and Robert Wilensky. Multivalent Annotations. In *Proceedings of the First European Conference on Research and Advanced Technology for Digital Libraries*, pages 287–303. Springer-Verlag, London, UK. 1997.

[QKH03] Dennis Quan, David R. Karger, and D Huynh. RDF Authoring Environments for End Users. In *Proceedings of Semantic Web Foundations and Application Technologies 2003*. Nara, Japan. 2003.

[RB01] Erhard Rahm and Philip A. Bernstein. A Survey of Approaches to Automatic Schema Matching. VLDB Journal, volume 10, no. 4; pages 334–350. 2001.

[RBG02] Nicholas Routledge, Linda Bird, and Andrew Goodchild. UML and XML schema. In *ADC '02: Proceedings of the thirteenth Australasian database conference*, pages 157–166. Australian Computer Society, Inc., Darlinghurst, Australia, Australia. 2002.

[RDF04] RDF Vocabulary Description Language 1.0: RDF Schema. Internet http://www.w3.org/TR/rdf-schema/. Accessed 30 Jan 2007. 2004.

[RDSM02] Allen Renear, David Dubin, and C. M. Sperberg-McQueen. Towards a Semantics for XML Markup. In *DocEng '02: Proceedings of the 2002 ACM Symposium on Document Engineering*, pages 119–126. ACM Press, New York, NY, USA. 2002.

[RDSMH03] Allen Renear, David Dubin, C. Michel Sperberg-McQueen, and Claus Huitfeldt. XML Semantics and Digital Libraries. In *Proceedings on Joint Conference on Digital Libraries*, pages 303–305. 2003.

[Rev88] György Revesz. *Lambda-Calculus: Combinators, and Functional Programming*. Number 4 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. 1988.

[Rie00] Doug Riecken. Introduction: Personalized Views of Personalization. Communications of the ACM, volume 43, no. 8; pages 26–28. 2000.

[RMRD05] Jolita Ralyté, Neil A. M. Maiden, Colette Rolland, and Rébecca Deneckère. Applying Modular Method Engineering to Validate and Extend the RESCUE Requirements Process. In Delcambre et al. [DKM+05], pages 209–224. 2005.

[ROH05] Lloyd Rutledge, Jacco van Ossenbruggen, and Lynda Hardman. Making RDF Presentable: Integrated Global and Local Semantic Web Browsing. In Allan Ellis and Tatsuya Hagino (editors), *WWW*, pages 199–206. ACM. 2005.

[RP06] Peter Rechenberg and Gustav Pomberger (editors). *Informatik Handbuch*. Hanser, 4 edition. 2006.

[RR98] Sudha Ram and V. Ramesh. Collaborative Conceptual Schema Design: A Process Model and Prototype System. ACM Transactions on Information Systems, volume 16, no. 4; pages 347–371. 1998.

[RSG01] Gustavo Rossi, Daniel Schwabe, and Robson Guimaraes. Designing Personalized Web Applications. In *Proceedings of the tenth international conference on World Wide Web*, pages 275–284. ACM Press. 2001.

[Rup04] Chris Rupp. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*. Carl Hanser Verlag, 3 edition. 2004.

[SAA99] Guus Schreiber, Hans Akkermans, and Anjo Anjewierden. *Knowledge Engineering and Management: The Common KADS Methodology*. MIT Press. 1999.

[SBS05] Hans-Werner Sehring, Sebastian Bossung, and Joachim W. Schmidt. Active Learning By Personalization – Lessons Learnt from Research in Conceptual Content Management. In J. Cordeiro, V. Pedrosa, B. Encarnaca, and J. Filipe (editors), *Proceedings of the 1st International Conference on Web Information Systems and Technologies*, pages 496–503. INSTICC Press. 2005.

[SBS06] Hans-Werner Sehring, Sebastian Bossung, and Joachim W. Schmidt. Content is Capricious: A Case for Dynamic Systems Generation. In *Advances in Databases and Information Systems, 10th East European Conference, ADBIS 2006*, volume 4152 of *LNCS*, pages 430–445. Springer Verlag. 2006.

[SCC05] Veda C. Storey, Roger H. L. Chiang, and G. Lily Chen. Ontology Creation: Extraction of Domain Knowledge from Web Documents. In Delcambre et al. [DKM+05], pages 256–269. 2005.

[SCD+97] Veda C. Storey, Roger H. L. Chiang, Debabrata Dey, Robert C. Goldstein, and Shankar Sudaresan. Database Design with Common Sense Business Reasoning and Learning. ACM Transactions on Database Systems, volume 22, no. 4; pages 471–512. 1997.

[Sch24] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. Mathematische Annalen, pages 305–316. 1924.

[Sch77] Joachim W. Schmidt. Some High Level Language Constructs for Data of Type Relation. ACM Transactions on Database Systems, volume 2, no. 3; pages 247–261. 1977.

[Sch99] A. Scharl. A Conceptual, User-Centric Approach to Modeling Web Information Systems. In *Proceedings of the 5th Australian World Wide Web Conference*, pages 33–49. Ballina. 1999.

[Sch06] J.W. Schmidt. Persistent Denotations for Conceptual Content Management: Foundations of a Content Calculus. Technical Notes. Personal communication. 2006.

[Seh04] Hans-Werner Sehring. *Konzeptorientiertes Content Management: Modell, Systemarchitektur und Prototypen*. Ph.D. thesis, Techische Universität Hamburg-Harburg. 2004.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. IEEE Software, volume 20, no. 5; pages 19–25. 2003.

[SGU02] Veda C. Storey, Robert C. Goldstein, and Harald Ullrich. Naive Semantics to Support Automated Database Design. IEEE Transactions on Knowledge and Data Engineering, volume 14, no. 1; pages 1–12. 2002.

[SGW05] Sebastian Schaffert, Andreas Gruber, and Rupert Westenthaler. A Semantic WIKI for Collaborative Knowledge Formation. In *Proceedings of SEMANTICS 2005 Conference*, pages 188–202. Trauner Verlag, Vienna, Austria. 2005.

[SHZ04] Yannis Smaragdakis, Shan Shan Huang, and David Zook. Program Generators and the Tools to Make Them. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 92–100. ACM Press. 2004.

[Sim94] Gary F. Simons. Conceptual Modeling versus Visual Modeling: A Technological Key to Building Consensus. Computers and the Humanities, volume 30, no. 4; pages 303–319. 1994.

[SM99] Frank M. Shipman III and Catherine C. Marshall. Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems. Computer Supported Cooperative Work, volume 8, no. 4; pages 333–352. 1999.

[SMBP] Sabrina Schönhart, Armin Müller, Laszlo Böszörmenyi, and Stefan Podlipnig. People behind Informatics - Virtual exhibition in memory of Ole-Johan Dahl, Edsger Wybe Dijkstra and Kristen Nygaard. Internet http://cs-exhibitions.uni-klu.ac.at/index.php?id=185. Accessed 25 Oct 2006.

[SMHR03] C. Michel Speerenberg-McQueen, Claus Huitfeldt, and Allen Renear. Meaning and Interpretation of Markup. Markup Languages: Theory & Practice, volume 3, no. 2; pages 215–234. 2003.

[SMJ02] Peter Spyns, Robert Meersman, and Mustafa Jarrar. Data Modelling versus Ontology Engineering. SIGMOD Record, volume 31, no. 4; pages 12–17. 2002.

[SMK02] Stefano Spaccapietra, Salvatore T. March, and Yahiko Kambayashi (editors). *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7-11, 2002, Proceedings*, volume 2503 of *Lecture Notes in Computer Science*. Springer. 2002.

[SMMS02] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-Driven Ontology Evolution Management. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 285–300. Springer-Verlag, London, UK. 2002.

[Som00] Ian Sommerville. *Software Engineering.* Addison-Wesley. 2000.

[Sow00] John Sowa. *Knowledge Representation.* Brooks/Cole. 2000.

[SPK+05] Constantine D. Spyropoulos, Georgios Paliouras, Vangelis Karkaletsis, Dimitrios Kosmopoulos, Ioannis Pratikakis, Stavros J. Perantonis, and Basilios Gatos. BOEMIE: Bootstrapping Ontology Evolution with Multimedia Information Extraction. In *Integration of Knowledge, Semantics and Digital Media Technology, 2005. EWIMT 2005*, pages 419– 420. 2005.

[SR02] Christiane Schmitz-Rigal. *Die Kunst offenen Wissens, Ernst Cassirers Epistemologie und Deutung der modernen Physik*, volume 7 of *Cassirer-Forschungen.* Ernst Meiner Verlag. 2002.

[SR03] Paul Shabajee and Dave Reynolds. What is Annotation? A Short Review of Annotation and Annotation Systems. Technical Report 1053, Graduate School of Education and Institute for Learning and Research Technology, University of Bristol, Bristol, UK. 2003.

[SS99] Joachim W. Schmidt and Hans-Werner Sehring. Dockets: A Model for Adding Value to Content. In Akoka et al. [ABCWM99], pages 248–262. 1999.

[SSS04] York Sure, Steffen Staab, and Rudi Studer. On-To-Knowledge Methodology (OTKM). In Steffen Staab and Rudi Studer (editors), *Handbook on Ontologies*, pages 117 – 132. Springer Verlag. 2004.

[SSSS01] Steffen Staab, Rudi Studer, Hans-Peter Schnurr, and York Sure. Knowledge Processes and Ontologies. IEEE Intelligent Systems, volume 16, no. 1; pages 26–34. 2001.

[SSW01] Joachim W. Schmidt, Hans-Werner Sehring, and Martin Warnke. Der Bildindex zur Politischen Ikonographie in der Warburg Electronic Library – Einsichten eines interdisziplinären Projektes. In Hedwig Pompe and Leander Scholz (editors), *Archivprozesse. Die Kommunikation der Aufbewahrung*, pages 238–268. Dumont. 2001.

[Sta03] International Organization for Standardization. Database Languages SQL, ISO/IEC 9075- *:2003. 2003.

[Sto77] Joseph E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The MIT Press. 1977.

[Sub96] Kazimierz Subieta. Object-Oriented Standards: Can ODMG OQL be Extented to a Programming Language? In *Proceedings of the International Symposium on Cooperative Database Applications*, pages 459–468. 1996.

[SWBM89] Joachim W. Schmidt, Ingrid Wetzel, Alexander Borgida, and John Mylopoulos. Database Programming by Formal Refinement of Conceptual Designs. IEEE Data Engineering Bulletin, volume 12, no. 3; pages 53–61. 1989.

[TH05] Yannis Tzitzikas and Jean-Luc Hainaut. How to Tame a Very Large ER Diagram (Using Link Analysis and Force-Directed Drawing Algorithms). In Delcambre et al. [DKM+05], pages 144–159. 2005.

[TMP98] R. Chung-Man Tam, David Maulsby, and Angel R. Puerta. U-TEL: A Tool for Eliciting User Task Models from Domain Experts. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces*, pages 77–80. ACM Press, New York, NY, USA. 1998.

[Tri05] Bill Trippe. Component Content Management in Practice. Technical report, X-Hive. 2005.

[Tro03] Raphaël Troncy. Integrating Structure and Semantics into Audio-visual Documents. In D. Fensel, K. Sycara, and J. Mylopoulos (editors), *The International Semantic Web Conference - Proceedings ISWC'03*, volume 2870 of *Lecture Notes in Computer Science*, pages 566–581. Springer Verlag. 2003.

[TW86] Randall H. Trigg and Mark Weiser. TEXTNET: A Network-Based Approach to Text Handling. ACM Transactions on Information Systems, volume 4, no. 1; pages 1–23. 1986.

[Uri05] José Alejandro Uriza Zepahua. *Conceptual Content Markup: A Prototypical Implementation*. Master's thesis, Technische Universität Hamburg-Harburg. 2005.

[VFC05] David Vallet, Miriam Fernández, and Pablo Castells. An Ontology-Based Information Retrieval Model. In Asunción Gómez-Pérez and Jérôme Euzenat (editors), *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 455–470. Springer. 2005.

[W3C99] W3C. XML Path Language (XPath). Internet http://www.w3.org/TR/xpath. Accessed 15 Dec 2006. 1999.

[W3C02] W3C. The Extensible HyperText Markup Language. Internet http://www.w3.org/TR/xhtml1/. Accessed 18 Dec 2006. 2002.

[W3C05a] W3C. Scalable Vector Graphics (SVG) Full 1.2 Specification. Internet http://www.w3.org/TR/SVG12/. Accessed 15 Dec 2006. 2005.

[W3C05b] W3C. Synchronized Multimedia Integration Language (SMIL 2.1). Internet http://www.w3.org/TR/2005/REC-SMIL2-20051213/. Accessed 10 Jan 2007. 2005.

[Wad02] Philip Wadler. XQuery: A Typed Functional Language for Querying XML. In Johan Jeuring and Simon L. Peyton Jones (editors), *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 188–212. Springer. 2002.

[WAS06] Vincent Wade, Helen Ashman, and Barry Smyth (editors). *Adaptive Hypermedia and Adaptive Web-Based Systems: 4th International Conference, AH 2006, Dublin, Ireland, June 21-23, 2006, Proceedings*, volume 4016 of *Lecture Notes in Computer Science*. Springer. 2006.

[Wat02] Dennis G. Watson. Brief History of Document Markup. Circular 1086, Agricultural and Biological Engineering Department, University of Florida. 2002.

[WC00] Kiri Wagstaff and Claire Cardie. Clustering with Instance-level Constraints. In Pat Langley (editor), *International Conference on Machine Learning*, pages 1103–1110. Morgan Kaufmann. 2000.

[Web90] *Webster's Desk Dictionary of the English Language*. Portland House, New York. 1990.

[Wed00] Lex Wedemeijer. Defining Metrics for Conceptual Schema Evolution. In Herman Balsters, Bert O. de Brock, and Stefan Conrad (editors), *FMLDO*, volume 2065 of *Lecture Notes in Computer Science*, pages 220–244. Springer. 2000.

[Wes04] Thijs Westerveld. *Using Generative Probabilistic Models for Multimedia Retrieval*. Ph.d. thesis, University of Twente, Enschede, The Netherlands. 2004.

[Wie92] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. IEEE Computer, volume 25; pages 38–49. 1992.

[WMBS04] Felix Weigel, Holger Meuss, Francois Bry, and Klaus U. Schulz. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In *Proceedings of Advances in Information Retrieval: 26th European Conference on IR Research, ECIR 2004*, volume 2997 of *Lecture Notes in Computer Science*, pages 378 – 393. Springer Verlag. 2004.

[WS03] Raymond K. Wong and Jason Sankey. On Structural Inference for XML Data. Technical report, University of New South Wales. 2003.

Index

λ-calculus, 21

abduction, 114
abstraction, 46, 64
abstraction axis, 71
abstraction variable, 22
ADL, *see* Asset Definition Language
$AE_{\rightarrow}$, 59
AE_C, 102
AEL, *see* Asset Expression Language
AEQL, *see* Asset Expression Query Language
AES, *see* Asset Expression System
annotation, 30, 153
    conceptual, 31
Any type, 60
application, 47, 64
application axis, 71
Asset, 6, 17
Asset class, 18, 102
Asset Definition Language, 18
Asset Expression, 44
Asset Expression Language, 43
    grammar, 48, 81
Asset Expression Query Language, 69
Asset Expression System, 85
audio, 57
axis, 70

bind, 46

callability, 116
cardinality, 115
Cassirer, 3, 4, 17, 149
CCM, *see* Conceptual Content Management
CCMS, *see* Conceptual Content Management System
class-based type system, 102
collaboration, 90, 143
component, 51, 74, 110
Conceptual Content Management, 18, 101
Conceptual Content Management System, 19, 145
content, 3, 44, 45
content axis, 71
context, 43
contraction, 114
conversion, 108
currying, 46

Delphi method, 155
demand, 116
denotational semantics, 23
design principles, 149
domain expert, 127
dynamics, 19, 81, 102

explanation, 47
expressions axis, 71
extensions of man, 149

firstness, 5

give-up, 117
GKNS, 126, 139

handling, 74
heterogenous content, 58
HTML, 56, 58
hypermedia, 154

iconographical, 5, 157
iconological, 5, 157
image, 57
impedance mismatch, 16
implementation, 137
inspector, 9, 90
intensional type, 103

keep, 117
knowledge base, 114
knowledge representation system, 114
KRS, *see* knowledge representation system

lambda-calculus, 21
lifting, 62, 64, 67
literal, 60
LLD, *see* low level descriptor
low-level descriptor, 57

map_C^{-1}, 107

map_S^{-1}, 107
map_C, 107
map_S, 104, 107
markup, 28, 153
    descriptive, 29
    generic, 29
    procedural, 28
    specific, 28
McLuhan, 149
mediator, 20
mention-handling, 74
modeling expert, 127
module, 19
MPEG-7, 35, 57, 153

name
    local, 91
named expression, 46
namespace, 91

object-oriented analysis, 155
On-To-Knowledge, 156
ontology, 24, 153
ontology engineering, 155
open modeling, 19, 81, 102
openness, *see* open modeling
operand, 22, 64
operand axis, 71
operator, 22, 64
operator axis, 71
OWL, 157

Panofsky, 5
parent axis, 71
partially applied expression, 81
path, 72
PDF, 58
Peirce, 5
penalty, 117, 119
personalization, 19, 81, 126
piece-handling, 75
plain text, 55
pre-iconographical, 5, 157
predicate, 72
prospector, 9, 90
prototype, 134

RDF, 157
rebind, 49, 64
reduction, 22, 75
relation, 13
relational model, 13
remove, 49
requirements analysis, 127

schema construction, 130
schema inference, 131
schema quality, 132, 136
secondness, 5
selector, 51
semantic type, 60
    description logic model, 115
    discovery, 113
    examples, 141
semantic web, 154
semi-structured data, 15
SGML, 29, 56
signature matching, 113
signature model, 114
SMIL, 59
specification, 136
structured document, 56
substitution, 22, 76
supply, 116
SVG, 57
symbol, 4

thridness, 5
trait, 67
    examples, 144
type construction, 61

UCSCP, *see* User Centric Schema Creation Process
usage, 137
use-handling, 75
User Centric Schema Creation Process, 125, 155

variable axis, 71
video, 58
visual notation, 48, 62

workspace, 89

XInclude, 39
XML, 57, 130
XML Schema, 38, 130
XPath, 40, 55, 56
XQuery, 39