INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Content Warehouses

Gunar Fiedler, Andreas Czerniak, Dirk Fleischer,
Heye Rumohr, Michael Spindler, Bernhard
Thalheim

Bericht Nr. 0605
March 2006

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Content Warehouses

Gunar Fiedler, Andreas Czerniak, Dirk Fleischer, Heye
Rumohr, Michael Spindler, Bernhard Thalheim

Bericht Nr. 0605
March 2006

e-mail: {fiedler,thalheim}@is.informatik.uni-kiel.de,
acz@informatik.uni-kiel.de,
{dfleischer,mspindler}@ipoe.uni-kiel.de,
hrumohr@ifm-geomar.de

Abstract

Nowadays, content management systems are an established technology. Based on the experiences from several application scenarios we discuss the points of contact between content management systems and other disciplines of information systems engineering like data warehouses, data mining, and data integration. We derive a system architecture called "content warehouse" that integrates these technologies and defines a more general and more sophisticated view on content management. As an example, a system for the collection, maintenance, and evaluation of biological content like survey data or multimedia resources is shown as a case study.

Contents

1 Introduction

1.1 Case Study: A Bio-Ecological Information System

Modern ecological research attempts to measure and explain global dependencies and changes and therefore needs to access and evaluate scientific data just in time. Nowadays, biological data is raised in an increasing number of experiments and surveys. A single biological experiment generates about 1.000 records while modern instruments and methods enhance quality of data.
Biological surveys are usually descriptive procedures. In a certain area of interest geographical referenced samples are taken, analyzed by a number of different methods, and probably compared with former surveys. In addition to invasive methods visual survey procedures were established in the last decades. Research projects that described regional communities of species produced large sets of so called survey data throughout the years.
Geographical referenced data sets facilitate the production of dissemination maps of species or higher taxa. Using the recorded data enables stock estimations. Hypotheses based on descriptive survey data need statistical foundations based on experimental data. Experiments are necessary to substantiate or reject a priori hypotheses.
Unfortunately, there is no general infrastructure for a free and uncomplicated exchange of data between researches as well as for public access to results of biological research. Publications in scientific journals suffer from multiple diseases. Published results do not include raw data, so it is necessary to get directly in contact with the author. The public is excluded, so scientific instiutions have to make additional efforts for public relations.
Beside the newly created data there are many historical sources. These data sets are often not available to the public in an electronic way. Many research projects as well as diploma or doctoral theses spend a lot of time (and money) to raise new data or to digitize historical data. Most of these projects are limited in time, the results are usually forgotten after a couple of years.
These problems are known since several years. To enhance availablitity and durability of biological data researchers are asked to store their results in the *Publishing Network for Geoscientific & Environmental Data* (PANGAEA, see [PAN06]). PANGAEA is a library of already published data. Unfortunately, diploma theses and other student work is not considered as publishable data anf therefore ignored. Additionally, transfering data into PANGAEA is a challenge for research projects. PANGAEA does not consider evaluation or presentation of data sets.

Some other projects try to address the topics mentioned above for restricted areas of interest:

- *Fishbase* (`http://www.fishbase.org`) is an information system with key facts about fish species.
- *ReefBase* (`http://www.reefbase.org`) is an online information system with facts about coral reefs.
- *AlgaeBase* (`http://www.algaebase.org`) is a taxonomic information system. The project supplies pictures of different algae.
- *Integrated Taxonomic Information System (ITIS)* (`www.itis.usda.gov`) is a database with taxonomic information of terrestrial and marine habitats.
- *SeaMountsOnline* (`seamounts.sdsc.gov`) transfers capabilities from information systems for terrestrial ecology to systems for marine ecology.
- *Ocean Biogeographic Information System* (`www.iobis.org`) is a component of "Census of Marine Life"; a network of institutions from 45 countries that investigates different parameters of marine organisms.
- *Meeresumwelt Datenbank (MUDAB)* (`www.bsh.de/de/Meeresdaten/Umweltschutz/MUDAB-Datenbank/index.jsp`) is the central database for environmental data from the North Sea as well as the Baltic Sea.

Analyzing the situation of modern biological research the need for a generalized repository for storing biological data sets emerges. The following observations can be made:

- The system should support different types of biological data with a focus on surveys and experimental data. Although there is a widely accepted aggreement on the general structure of data, the system should be flexible enough to facilitate different facets of different topics in biological research.
- Data sets have to be syntactically and semantically annotated and documented. Documentation of failed experiments is as important as the publication of successful experiments to facilitate collective learning. Usage of raw data should be traceable: Who used the data? Which publications are based on these data sets? Which experiments are correlated with a certain survey?
- The system should be able to supply a browseable (e.g. via a taxonomy) and searchable archive of multimedia data such as pictures and video sequences, e.g. as a surplus value for public relation management of biological research.

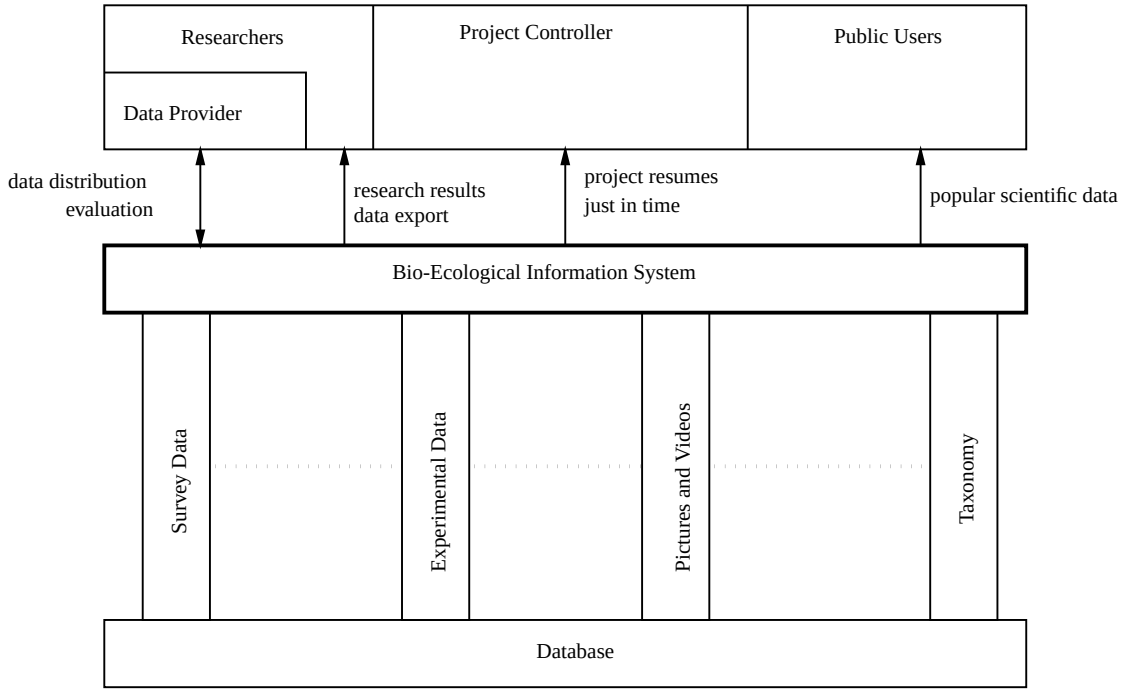


Figure 1.1: The Four Columns of a Bio-Ecological Information System

- There are a number of standard evaluation and visualization methods. The repository should enable data providers (researches who provide data sets within the system) to use these evaluation and visualization techniques.
- The repository acts as a proxy for other biological information systems (e.g. PANGAEA). Data sets within the repository can be transfered to PANGAEA in a (semi-)automated way which makes it easier for the researcher to publish scientific results.

Figure 1.1 shows the four columns of a bio-ecological data repository: management of survey data, management of experimental data, management of multimedia elements and management of taxonomical information. Different user groups with different needs can be identified:

- *Data providers*, supplying data sets for the repository are interested in evaluating and visualizing these data sets according to standard methods. Additionally, the data providers are disburdened of the responsibility for long-term storage of their data sets. This enables data sharing within globally distributed virtual working groups because every participant can access the data of the working group from everywhere at every time.
- *Reseachers* are interested in finding data sets for comparative studies. To prevent uncontrolled access and misuse of data the system has to implement

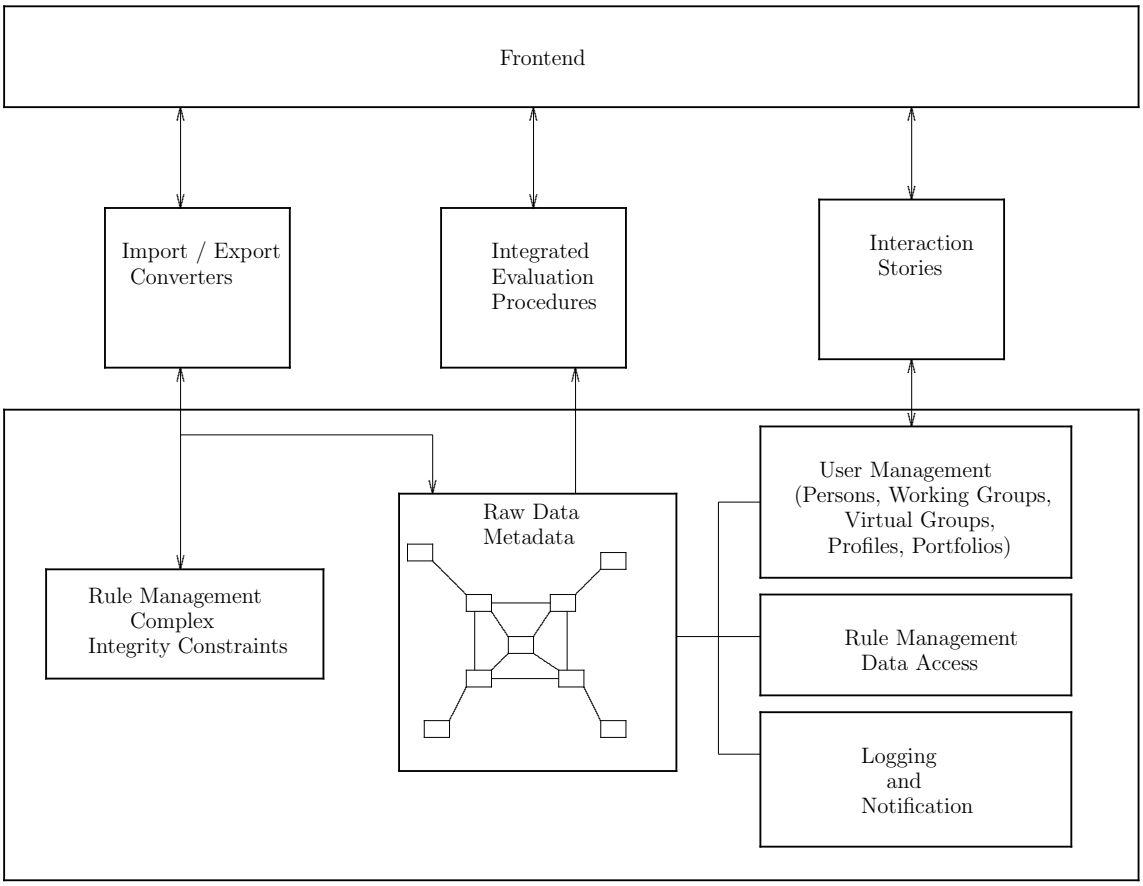


Figure 1.2: System Components

the "rules for a good scientific practice", e.g. a retention period for newly raised data sets and a notification of the data owner if sets are exported.

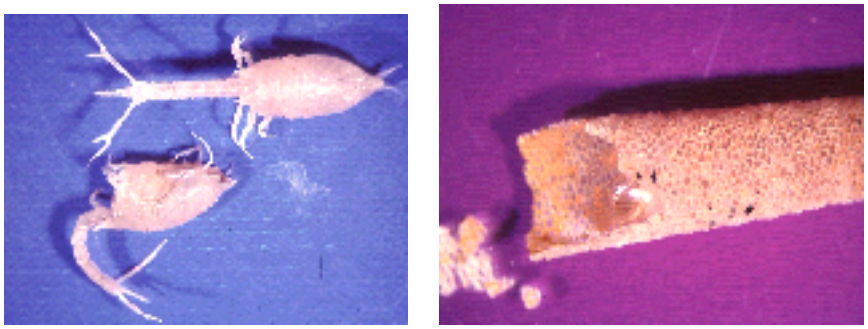
- *Project Controllers* are able to participate in the project's evolvement from the beginning. Statistical resumes can be derived from the data within the repository.

- *Public users* are supplied with popular scientific reports, such as pictures, video clips, visualized statistics, annotated maps, resumes of research projects, or selected publications.

Figure 1.2 shows the main components of a bio-ecological information system from a technical point of view:

- Survey data and experimental data is stored within a central database. Each raw data item can be annotated with metadata such as authors, creation date, geographical reference, descriptions, citations, resumes, links to publications.

(a) Picture from a video clip (ground of the Baltic Sea, Kiel Bay)



(b) Detailed picture from *Cumace Diastylis* (left) and *Lagis koreni* (right)

Figure 1.3: Multimedia Data

Additionally, pictures and video clips are stored together with taxonomic information, figure 1.3 shows examples. The schema of the central data repository has to be flexible enough to cope with special facets of different surveys or experimental methods. At the same time the schema has to carry enough information to represent the data's semantics in an appropriate manner. To ensure a certain quality level complex integrity constraints have to be defineable. Semantics of these rules should be represented in the repository, not only in the application's code to ensure understandable and changeable import policies. Constructing flexible structured schemas is discussed more deeply in chapter 2.

- To support virtual working groups the information systems needs powerful user management facilities. Definition of real and virtual groups is necessary as well as the explicit definition of persons, user profiles (the user's characteristic properties which can be used to deduct needs, wishes, and possibilities of users) and portfolios (tasks and plans of the user). It must be possible to dynamically define access permissions based on groups, profiles and portfolios as well as scientific rules (e.g. retention periods or the publication state of certain results). Chapter 3 discusses this topic.

- Access to raw data has to be logged because of security reasons. To prevent misuse of data each data provider has the right to be notified in a configurable way if its data sets are exported. The information system produces statistical evaluations on the usage of data sets, e.g. for project resumes.

- An important part of the bio-ecological information system is the integration of standard evaluation methods such as Bray-Curtis Similarity Matrices, the Simpson Index, the Margalef Index, the Shannon-Weaver Index, Pielou Evenness, Taxonomic Distinctness, AMBI, BQI, ANOVA, ANOSIM, or statistical significance tests. Results are visualized, figure 1.4 shows an example.

- Each user group (e.g. researcher, data provider, public user) has certain needs and wishes according to the personal profile. Adaptable user interfaces allow an ergonomic usage of the system's functionality. Chapter 5 discusses the topic of modeling user interfaces.

- Data providers import and export data sets. The repository has to support different data exchange formats. Additionally, the repository should support data exchange with existing biological archives like PANGAEA. The mapping between different schemata and formats (both on a syntactical and semantical level) and cooperation between information systems is covered in chapter 4.

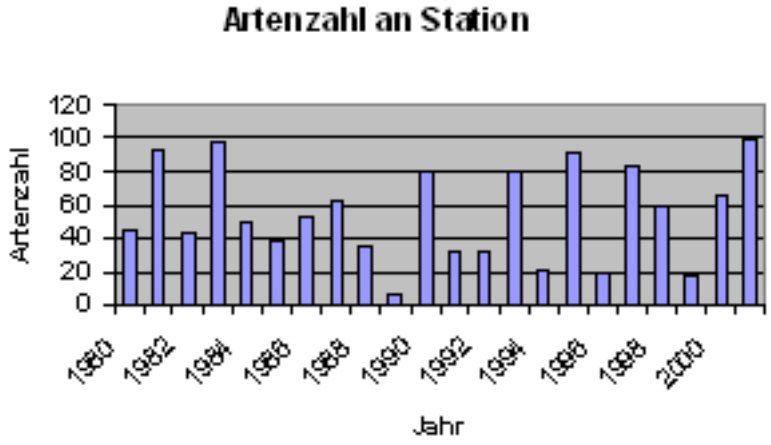


Figure 1.4: Example of Data Visualization: number of species over a longer period of time

1.2 From Content Management Systems to Content Warehouses

The architecture of an bio-ecological information system discloses many points of contact to different research topics in information system engineering, such as content management systems, data mining technologies, semantic databases, and modeling issues. It is typical for a whole family of information systems with a broad variety of application areas. We will call these kind of information system *Content Warehouse*. The detected requirements can be generalized and applied to different areas of interest. The following sections and chapters provide an overview over related technologies and actual research directions.

1.3 Related Work

1.3.1 Content Management Systems

Content Management Systems (CMS) support administration and maintenance of sharing "content" within an organization. The term "content" denotes some piece of information. Usually, there is no clear separation between "content" and "information", but "content" is used in a more technical way as a piece of eletronically provided structured, semistructured, or unstructured data that is viewed by users, changed by editors, or shared between users or organizations. Many (Web) content management systems (WCMS) restrict "content" to the contents of web pages with reusable static or dynamic chunks of data (called assets), e.g. text fragments, pictures, movie clips, links, data from databases, or applets.

An important design principle of content management systems is the separation between assets (raw data), structure, and presentation. Assets are combined with meta data (information about author, creation date, keywords, usage context, versions, access rights, payment status, etc.). Editors create (semi-)structured documents by assembling assets, usually by using a structural template. Structural templates are associated with visual templates (e.g. JSPs or PHP scripts) that manage visualization of content documents. It is possible to manage different visualization templates for structural ones to facilitate different output channels (e.g. HTML pages, WML output for WAP enabled cellphones, PDF for print media, or VoiceXML for voice based interaction.)
The main goal of content management is the automation of processing reusable content. For that reason the whole content life cycle is supported:

- *Content creation* and *content import:* assets are managed according to definable design rules. Editors can use document templates for creating structured documents. Usually, there exist converter modules for importing content from external sources based on standard exchange formats.

- *Content maintenance:* content is associated with metadata, such as author information, usage context, or keywords. This supports editors in finding content for reuse. Versioning of content is usually supported.

- *Content distribution:* content can be distributed in two different ways: structured for reuse in other contexts or rendered for presentation. Exchange and linkage of content between organizations (or between different web sites) is called *content syndication*, e.g. on the basis of RSS, NITF[1], or legacy formats.

- *Content monitoring:* access to certain content is restricted to the user's profile and portfolio. Which tasks the user has to or wants to fulfill? Is the content classified as internal content or public content? Is there any need to pay for accessing the content? Is the content actually under processing in a certain workflow?

There are three different types of content management systems:

- *Document oriented systems* are designed to manage unstructured or weakly structured bulk documents, e.g. articles or books in an electronic library. These systems usually supply information retrieval methods and conversion between different formats such as PDF or PostScript.

- *Workflow oriented systems* focus on workflows or document flows. The main goal is the support of business rules and business processes together with personal views on processes and data. Workflow oriented systems are widely used in B2B applications [2] or distributed information systems.

[1]News Industry Text Format
[2]business to business applications

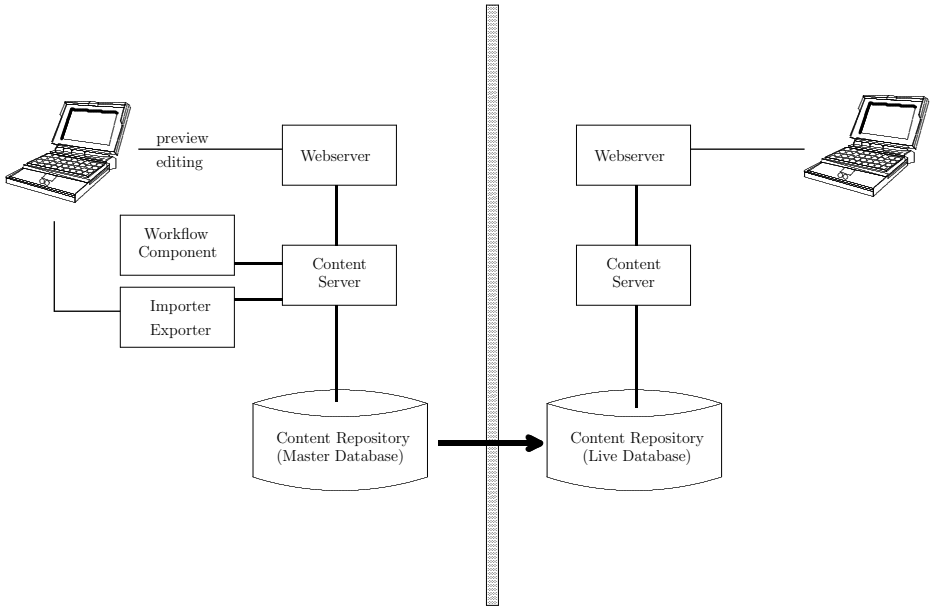


Figure 1.5: Architecture of Content Management Systems

- *Editorial systems* are used for creating newspapers or websites. Reuse of produced or purchased content in different contexts and on different medias is supported. Typical workflows known from the editorial business (such as "four-eye-publication" and others) are usually supported but often not in a generic way.

A typical system architecture of a content management system is depicted in figure 1.5. For security and performance reasons the system is typically divided into two parts: the master system and the live system. The master system (also called "backend system") is used by the editors to create and maintain content. It is usually accessible within the intranet of the organization and protected against access from the extranet. Content is stored in a repository which is usually a relational database or a collection of files in a file system. The content application server is the central component of the content management system controlling the access to the content within the repository. Optionally, the system includes a workflow component for controlling the state of content objects within business processes. Content is created and maintained by editoral and administrative applications. These are either Web based interfaces or proprietary applications. For supporting content syndication or changing content in external applications (e.g. editing pictures in professional graphic software) content import and export modules are usually present.
The second part of a content management system is the live system (also called "frontend system"). It supports the efficient retrieval of content. It is also backed by a content repository. This repository contains a subset of the data from the master content repository, namely the published content. The process of transfering data

from the master system to the live system is called publication. Publication takes place when an editor triggers the publication process or at regular times to prevent unauthorized modifications of live content from outside.

1.3.2 Data Warehouses

Introduction

Operational data in enterprises is often distributed over several specialized and probably isolated applications. Every Application stores relevant data in a locally optimized way. Even if there is only one application, data is usually not structured for analysis. This often prevents or complicates a global view on the data as a base for sound management decisions or recognition of dependencies and coherences.
The concept of data warehouses deals with this situation. A data warehouse is a central database which collects data (so called micro data) from different sources. Micro data is transformed, integrated and supplied for evaluation. In difference to specialized applications, a data warehouse facilitates a global, rectified, and cleaned view on the collected data of the whole enterprise.
Data warehouse systems consist of a complex network of cooperating components. The core of a data warehouse is a central database. The separation of operational data and data evaluation is one of the central aspects of data warehousing because applications are not influenced in terms of performance. On the other side, evaluation tools with complex and long-running queries need a completely different data organization. Because of the historical dimension data warehouses store significantly more data than necessary for the actual work.
Data from local applications is periodically (or even just in time) loaded into the data warehouse. This procedure separates data warehouses from logically integrated federated database systems. Federated databases also allow a global view on a network of databases as well as global processing, but queries are delegated to the local systems.
Beside data storage the loading process is a challenge for data warehouses. Different data sources supply data with different granularity and quality. The data warehouse has to transform and to integrate this mix of data. It has to complete missing data in a meaningful way, recognize overlapping data, resolve conflicts and errors, filter irrelevant data, and calculate derived information from the raw data. Data sources deliver data in different formats, e.g. data from relational databases, hierarchical or network-based databases, XML documents, or legacy files.
The surplus value of a data warehouse for end-users is determined by available evaluation and transformation tools. There are report generators for generical summaries of data. Many statistical tools enable the verification of hypotheses. OLAP[3] applications support the classification and evaluation of data defined over multiple dimensions. Data mining tools enable the user to formulate forecasts based on the data from the past and the present, e.g. the generation of hidden association rules.

[3]Online Analytical Processing

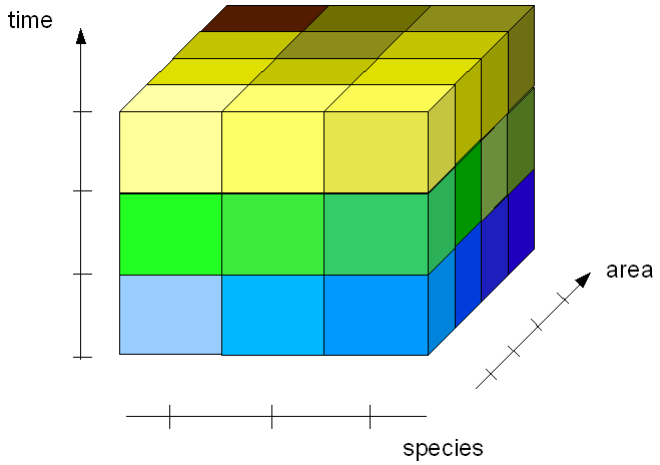


Figure 1.6: A data cube for biological surveys with three dimensions: time, area, and species. Each cell contains the corresponding number of individuals, denoted by different colors.

Other possible tools are data filters for personalized data (e.g. conversions between units), agents informing the user of certain events and constellations, visualization tools, or decision support systems. Most of these applications are restricted to predefined analysis scenarios but they are usable with only few initial training.

Online Analytical Processing

In OLAP applications data within a data warehouse is usually visualized as a multi-dimensional cube. Each cell within the cube contains the measures of interest.
Consider the following scenario: on regular surveys the number of individuals of certain species in certain areas is measured. We have three dimensions: time, area[4], and species. Each cell within this three-dimensional cube contains the number of individuals of the corresponding species those were found at this point of time on this coordinates. Figure 1.6 depicts this scenario.
For each dimension categories can be introduced, e.g. days, months, years, and decades for the time dimension, regions for the area dimension, or a taxonomy for species. Figure 1.7 shows an example.
The data cube supports summarization with groupings based on categories (or, more general, properties), e.g. the average number of individuals of a certain species over the last decade. Data cubes implement specialized operations for interactive data analysis, e.g.:

- *drill down:* the cube's cells are replaced (in a visual sense) by cells with a finer granularity (e.g. cells containing the number of vertebrates are replaced by cells containing the number of fishes, amphibia, reptiles, birds, and mammals.)

[4]Area is actually two-dimensional (longitude and latitude).

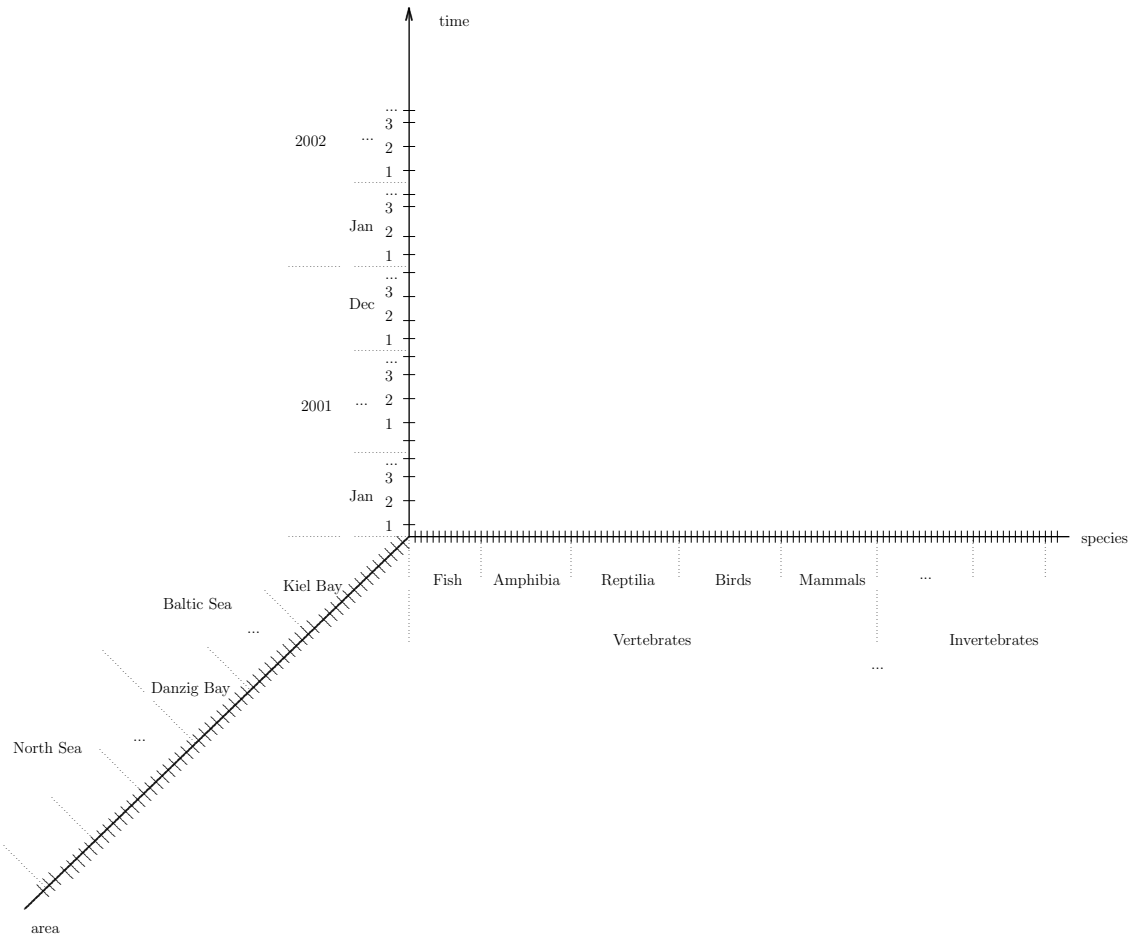


Figure 1.7: Categories for Cube Dimensions

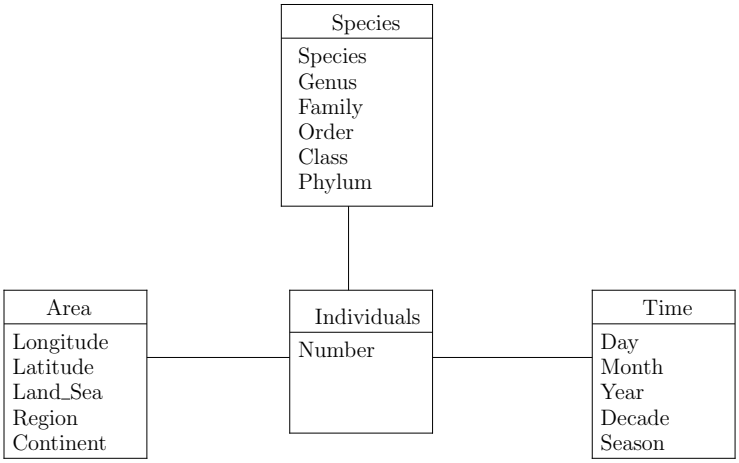


Figure 1.8: Star Schema

- *drill up:* cells are summarized (e.g. cells containing the average numbers of individuals in certain months are replaced by a cell containing the average number of individuals over the year. The operation *roll up* creates summaries on different levels of abstraction.

- *slice and dice:* sub cubes are created by applying filters (e. g. the user is only interested in data from the year 2004 recorded in the Baltic Sea concerning fishes.)

Data cubes are implemented using either multi-dimensional arrays (MOLAP) or relational structures (ROLAP). ROLAP databases usually consist of dimension tables and fact tables, containing the measured data and foreign keys to all dimension tables. Fact tables and dimension tables form a star-like schema as depicted in figure 1.8. If dimension tables are normalized or connected to other tables the star schema is called "snowflake schema".
Although traditional database systems can handle typical ROLAP queries, it is a matter of performance to represent the data cube in an appropriate way. A cube can be defined as a set of cube views over a relational (star or snowflake) schema $\mathcal{S}$. In principle, there are three different ways to handle cube queries:

- Each query is rewritten according to the view definition of the current cube configuration and is executed on the base relations. This approach is storage-efficient (only the base data has to be stored) but it is very time consuming.

- Cube views are completely materialized, all queries to a certain cube configuration is processed using these materialized cube views. Using this configuration the best query performance can be achieved but a lot of disk space is needed to hold the materialized views.

- Cube views are only partially materialized. Views that are not materialized are calculated using materialized views. This approach is a trade-off between space and time consumption.

1.3.3 Data Mining

Data Mining is the process of automatically discovering useful information in large data repositories ([TSK06]) and is a part of knowledge discovery. Data Mining tasks can be divided into two categories:

- *Prediction:* Dependent attributes are predicted based on a set of given attributes.
- *Description:* Descriptive procedures derive patterns that summarize hidden relationships in data.

Data mining processes usually consider data objects with attributes. An attribute corresponds to a property or characteristic of an object. Attributes may vary from object to object or over time. Attributes are associated with a (discrete or continous) domain of possible values. There are different kinds of domains:

- *nominal attributes:* sets of distinguishable elements, e.g. color or gender
- *ordinal attributes:* sets of distinguishable elements with an ordering relation, e.g. grades ($\{good, better, best\}$)
- *interval attributes:* there is a unit of measurement so differences between elements are meaningful, e.g. dates, temperatures
- *ratio attributes:* both differences and ratios are meaningful, e.g. length, mass

Based on values of attributes different techniques can be applied:

Classification assigns objects to predefined categories. Given a set A of attributes a classification model has to be established that maps objects with certain values for attributes from A to classes.
The input for the classification algorithm is a set of objects over the attribute set $A' = A \cup \{c\}$. where c is the target attribute, also called "class label". The classification algorithm employs a learing algorithm that identifies a classification model that best fits the underlying relationships between the attributes. Possible classification techniques include classification based on decision trees, rules, neural networks, or support vector machines.

Association analysis try to discover hidden association rules. An association rule is an implication of the form $A \rightarrow B$ where A and B are disjoint sets of values. A typical application is the "market basket" scenario: which products were bought together?

Clustering divides a set of objects to several groups. Usually, the clusters should capture some natural structure of the data. Dividing the world in conceptually meaningful clusters is used as a starting point for other purposes like data summarizations.

1.4 Logical Foundations of Content Warehouses

1.4.1 Separation of Syntax, Semantics, and Pragmatics

Content managed in content management systems as seen in section 1.3.1 is characterized by three dimensions:

- raw data combined with meta data stored in a data repository,
- structuring and relationships of data fragments based on a predefined content object model, and
- presentation instructions, usually consisting of stylesheet suites.

These dimensions can be found in any slightly generic CMS while the implementations differ from one system to another. Although the need for separation between these dimensions is very present, available systems lack of a conceptual and computational support for more sophisticated tasks. Structuring as well as presentation of content strongly depends on the content's usage; it depends on the user's profile and task portfolio. Different users need different "information units" to perform their associated tasks. These information units may have different or multiple structurings because they may be stored in or delivered by different independent sources. All this can be done using a traditional CMS by coding the logic to the view suite or to the presentation templates. But following the tradition of database management systems this functionality should be provided by the CMS in a more generic way. Additionally to the management of structures we need a management of logical descriptions to handle the content's semantics behind the data and a management of topics to deal with the pragmatic use of data.
Dealing with logical extensions to database systems is not new. Deductive databases (see [Min88]) combine logic programming and databases. The asset approach of content management systems ([Seh03; SS04]) uses conceptual containers (assets) for a subject-oriented access. "Semantic Web" technologies and languages such as the OWL Web Ontology Language deal with similar issues. Reasoning support for description logics based approaches is available (see e.g. [Sys; Hor98]).
By generalizing current approaches, we identify three different perspectives of a content management system, depicted in figure 1.9. The main difference is the strict separation between the notion "information" and the notion "content". Information is data that is perceived or noticed, it is selected and organized by its receiver with

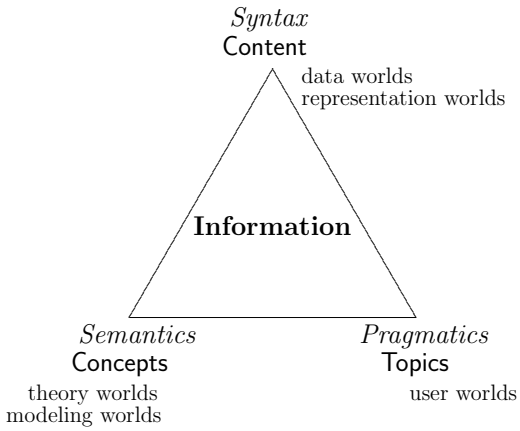


Figure 1.9: Semiotic Separation of Content, Concepts, and Topics

respect to his/her subjective interests and integrated in his/her recallable knowledge. That's why information is more than content: it is content combined with its interpretation and the understanding of its users.
As shown in figure 1.9 we use the term content in this restricted form: content means elements of business data. The notion "concept" is used for small logical theories describing the semantics. Different users may have their own specific terminology to denote content and concepts. We use the notion of "topics" for these denotations. So we can distinguish between three different worlds:

- Looking at the *content world* we concentrate our attention to data, data structuring, and data representation.
- Considering the *concept world* we investigate the logical foundations of the content world as well as the topic world.
- Looking at the *topic world* we are interested in the terminologies or ontologies of users and in their common understanding.

1.4.2 The Content World

The content world deals with content object suites. Content objects may be structured, semi-structured or unstructured. A suite is a set of objects combined with an integration or association schema and the obligations required for maintaining these associations. Content objects are based on a type system describing the structuring and the functionality of these objects. Associations are expressed by relationship types and constraints. The functionality of content objects is specified by a retrieval expression, a maintenance policy and a set of functions supporting the utilization of the content object and the content object suite. The notion used here extends modern approaches ([SS03]) and combines them with the theory of media objects

([FKST00]). Classically, (simple) views are defined as singleton types collecting data from the database by some query.

```
create view  name (projection variables)
 select       projection expression
   from       database sub-schema
   where      selection condition
 group by     expression for grouping
   having     selection among groups
 order by     order within the view
```

Simple examples of view suites are already discussed in [Tha00] where view suites are ER schemata. The integration is given by the schema. Obligations are based on the master-slave paradigm, i.e., the state of the view suite classes is changed whenever an appropriate part of the database is changed.
We generalize the view specification frame used in relational databases by the frame:

```
generate MAPPING :
            VARS → OUTPUT STRUCTURE from DATABASE TYPES
 where SELECTION CONDITION
 represent using GENERAL PRESENTATION STYLE
  & ABSTRACTION (GRANULARITY,
            MEASURE, PRECISION)
  & ORDERS WITHIN THE PRESENTATION
  & HIERARCHICAL REPRESENTATIONS
  & POINTS OF VIEW
  & SEPARATION
 browsing definition CONDITION
  & NAVIGATION
 functions SEARCH FUNCTIONS
  & EXPORT FUNCTIONS
  & INPUT FUNCTIONS
  & SESSION FUNCTIONS
  & MARKING FUNCTIONS
```

A content schema $\mathfrak{D}$ on a database schema $\mathcal{S}$ is given by a view schema $\mathfrak{D}_v$, a defining query q_v transforming databases over $\mathcal{S}$ into databases over $\mathfrak{D}_v$, and a set of functions defined on the algebra $\mathfrak{A}(\mathfrak{D})$ on the content schema. The defining query q_v may be expressed in any suitable query language, e.g. query algebra or SQL.
Looking at views in the common definition some concepts are missing, e.g. the concept of *links*. This can be achieved by introducing some kind of "objectification" of values in the query language. We should talk about *query languages with create-facility*.

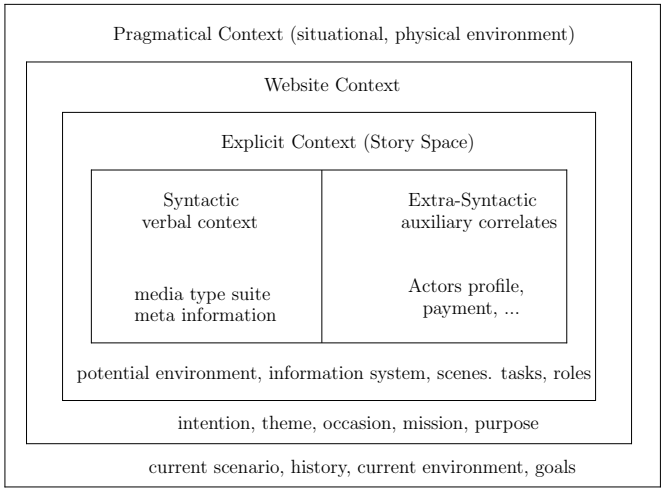


Figure 1.10: Wrappers for Content Types

Additionally, we want to integrate auxiliary content ([LRL98]) getting *extended content types* with a name $\mathfrak{D}^{\mathcal{X}}$, a content schema $\mathcal{D}$, a defining query $q_{\mathfrak{D}^{\mathcal{X}}}$ with create-facility and a binding between $\mathfrak{D}$ and $q_{\mathfrak{D}^{\mathcal{X}}}$.
The representation of content types and auxiliary content types depends on the users profile, the task portfolio, units of measures, presentation types, the order of presentation, and the container for delivery ([ST01]). Therefore content types have to be wrapped for representation to be adaptable to users and the environment, as depicted in figure 1.10.

1.4.3 Concepts as Small Logical Units

Objects, Properties, and Relationships

In the concept world we generalize the structure given in the content world and speak about objects. Let $\mathcal{O}$ be a set of abstract objects.
We define relationships between objects. Let $\mathcal{R}$ be a set of roles. A relationship b between objects is a total function that maps roles r from a set $\{r_1, ..., r_n\} \subseteq R$ to objects o from $\mathcal{O}$: $b : \{r_1, ..., r_n\} \rightarrow \mathcal{O}$.
Further, let $\mathcal{P}$ be a set of abstract properties. Each property $p \in \mathcal{P}$ is associated with a type from a set of types $\mathcal{T}$, denoted as $type(p)$, e.g. base types, tuples, sets, bags, or lists. Definition of types can be found e.g. in [Tha00]. Because types are more an issue in the content world we do not deepen the notion of types here. Typical properties of objects on the conceptual level are properties with a base type like boolean values, numbers, or strings.
Objects $o \in \mathcal{O}$ can be described by a set of properties. The function $prop$ maps each object o to a set of pairs (p, v) where p is a property from $\mathcal{P}$ and v is a value from $dom(type(p))$.

Concepts

In general, a concept is a tuple (m, int, ext) where ext (the concept's extension) is a set of objects, int (the concept's intension) is a logical formula describing the objects in the concept's extension by using propositions over the the object's properties, and m is a function associating the concept with descriptive meta data such as a name. If meta data is not of interest, m is omitted.
Let $\mathcal{K}$ be a set of abstract concepts. An extension function $\mathcal{E}$ is a function $\mathcal{K} \rightarrow \mathcal{O}$ which associates a set of objects to a concept. An intension is a function $\mathcal{I} : \mathcal{K} \rightarrow \mathcal{L}$ that associates each concept with a formula in a given language $\mathcal{L}$. As an abbreviation, we use $\mathcal{E}(\varphi)$ with a formula φ to denote the extension of the concept with the intension φ as well as $\mathcal{I}(\mathcal{A})$ with a set of object $\mathcal{A}$ to denote the intension of the concept with the extension $\mathcal{A}$.
Concepts can be categorized:

- *Extensional concepts* are defined by explicitly specifying the concept's extension. The concept's intension is derived.
- *Intensional concepts* are defined by specifying the concept's intension. The concept's extension is derived.
- *Mixed concepts* are defined by specifying intension and extension. The intension can be seen as a set of constraints that have to be fulfilled by all objects in the concept's extension. The concept's extension is partially defined. These explicit specifications are called *assertions*.

An inference machine within a conceptual content management system supports evaluation methods, e.g.

- Is concept $\mathfrak{C}_1$ subsumed by concept $\mathfrak{C}_2$ ($\mathfrak{C}_1 \sqsubseteq \mathfrak{C}_2$)?
- Which concepts subsumes a concept $\mathfrak{C}$?
- Are two concepts $\mathfrak{C}_1, \mathfrak{C}_2$ equivalent (or disjoint)?
- Is concept $\mathfrak{C}$ satisfiable?
- Is object o instance of concept $\mathfrak{C}$?
- Which objects are instances of concept $\mathfrak{C}$?

Applying techniques from Artifical Intelligence, additional functionality can be exposed, e.g.

- Generate a concept lattice for a given set of objects.
- Learn an intension for given sets of objects (learning set, verification set).

The conceptual world of content management systems can grow very fast. Even under the restriction of simple languages (e.g. description logic, see [BCM+03; HMW05]) efficient computational support can become a nightmare. To limit the number of relevant concepts and objects within a computation should be the first goal. In typical content management applications the concepts present in the repository are not completely connected to each other. There are clusters of concepts with strong relationships while these clusters are only loosly coupled. A concept $\mathfrak{K}_1$ from a cluster C_1 has only weak (or even no) influence on calculations on concepts $\mathfrak{K}_2, \mathfrak{K}_3$ from cluster C_2. For example, consider a CMS containing content for employees of a company. Concepts denoting spare time activities (members of the company's sailing team, etc.) are not relevant while querying organizational structures (e.g. all members of a certain department). On the other side, in some situations it can be meaningful to combine both concepts. Additionally, it is useful to allow different languages to express concepts in certain situations, e.g. description logic, temporal logic, epistemic logic, or others. These concepts should be compatible, but it is not a promising approach to integrate all languages in a common "super language".
To support these requirements the notion of *world views* is introduced. A world view $\mathcal{V}$ is a tuple $(\mathcal{L}^{\mathcal{V}}, \mathcal{U}_O^{\mathcal{V}}, \mathcal{B}^{\mathcal{V}}, \mathcal{C}^{\mathcal{V}}, prop^{\mathcal{V}}, \mathcal{I}^{\mathcal{V}}, \mathcal{E}^{\mathcal{V}})$ with

- $\mathcal{L}^{\mathcal{V}}$ is the language within this world view. For example, using $\mathcal{SHIQ}$ descripion logic provides expressiveness known from "Semantic Web" inference machines for the OWL Web Ontology Language, see e.g. [McA00; Sys; Hor98].
- $\mathcal{U}_O^{\mathcal{V}}$ is the world view's universe of objects (all abstract objects residing within this world view). $\mathcal{U}_O^{\mathcal{V}}$ is a subset of $\mathcal{O}$. There are different possibilities to construct $\mathcal{U}_O^{\mathcal{V}}$:
  - $\mathcal{U}_O^{\mathcal{V}}$ is determined by the extensions of all concepts within $\mathcal{V}$:
    $$o \in \mathcal{U}_O^{\mathcal{V}} \Leftrightarrow (\exists c)(o \in \mathcal{E}^{\mathcal{V}}(c))$$
  - $\mathcal{U}_O^{\mathcal{V}}$ is a finite set of objects. The objects within the concept's extensions have to be a member of $\mathcal{U}_O^{\mathcal{V}}$, but it is possible that there are additional objects not included in any extension. $\mathcal{U}_O^{\mathcal{V}}$ can be fixed or dynamic with respect to updates.
  - $\mathcal{U}_O^{\mathcal{V}}$ is potentially infinite. This restricts possible operations within this world view to prevent unsafe queries.
- $\mathcal{B}^{\mathcal{V}}$ is the set of relationships between objects in $\mathcal{U}_O^{\mathcal{V}}$.
- $\mathcal{C}^{\mathcal{V}}$ is the set of concepts within this world view.
- $prop^{\mathcal{V}}$ is a function that assigns properties to objects.
- $\mathcal{I}^{\mathcal{V}}$ is the intension function.

- $\mathcal{E}^{\mathcal{V}}$ is the extension function.

Within a world view reasoning takes place under the possibilities and restrictions of the chosen language. World views can be related to each other. A world view exposes a set of concepts while each concept $\mathfrak{C}$ is represented by a predicate $C(o)$ which is true if the given object is an element of $\mathfrak{C}$'s extension.
The notion of properties can also be applied to concepts. For a concept $\mathfrak{C}$ we can express

- which properties an object in the concept's extension must have,
- which properties an object in the concept's extension can have,
- which properties an object in the concept's extension must not have.

Additionally, default or fixed values for properties can be provided by the concept. In terms of the world view's interface the concept acts as a prototype of its objects. In a similar way we can attach relationships to concepts; and we can define concepts over relationships, concepts of word views, and concept templates (concepts depending on a number of parameters which are fixed at evaluation time).

1.4.4 The Topic World

Our topic notion generalizes and formalizes the notion of topics [Pe01] commonly used for topic maps and implemented in [Ont]. Topic maps are based on conceptual structures [Sow00]. Our notion integrates these proposals with the Pawlak information model [Paw73] and concept lattices [GW98]. A topic $\mathfrak{T}$ can be described by its *user characterization*, its *topic description* and its *topic population*.
For example, the topic of *address* specifies addresses by means of geographical addresses or contact addresses. Typical specific addresses are *living address*, *mailing address*, and *delivery address*. The address topic does neither cover *diplomatic addresses* nor *memory locations*.

2 Data Management

Database applications tend to be large, unsurveyable, incomprehensible, and partially inconsistent due to the applications, the database development life cycle and due to the number of team members involved at different time intervals. Thus, consistency management of the database schema might become a nightmare and may lead to legacy problems. The size of the schemata may be very large, e.g. the size of the SAP R/3 schema consisting of more than 21.000 tables. In contrast, [Moo01] discovered that diagrams quickly become unreadable once the number of entity and relationship types exceeds about twenty.
This chapter discusses modular schema design that supports schema evolution and schema adaptation by structuring and decoupling facets of the schema. Surveyability is enhanced by encapsulation and information hiding. Modular schema design is based on schema components: reusable and intermountable parts of the schema wraping certain facets. The discussion extends the approach in [Tha05].

2.1 Component Based Modeling

Large database schemata can be drastically simplified if techniques of modular modeling such as *design by units* ([Tha00]) are used. Modular modeling is an abstraction technique based on principles of hiding and encapsulation. Design by units allows to consider parts of the schema in a separate fashion.
The term "component" is widely used in different engineering disciplines. Software engineering ([HC01]) defines a software component as a piece of software conforming to a component model that can be combined with other elements according to a composition standard without any change. Various component models for different application areas have been developed throughout the years, e.g. JavaBeans, OLE, COM, OpenDoc. Additionally, it is often claimed that a component should be reusable in different application contexts and therefore it should be parameterizable. Components implementing the same "semantics" are claimed to be intermountable. To achieve these goals design pattern where defined (see e.g. [GHJV95]). Components are often organized in component libraries.
Similar approaches can be found in electrical engineering for "intellectual property libraries" defining reusable hardware components in a black-box style. Each component is specified by (input and output) ports, a behavioral description in a certain language (e.g. VHDL) and secondary meta data describing requirements for the component's context. This interface specification can be associated with different implementations which are hidden for the component's user.

2.1.1 Database Components

The approach in [Tha05] uses the extended ER model HERM defined in [Tha00] for representing structure and behavior. A database type $\mathcal{S} = (S, O, \Sigma)$ is given by

- a structure S defined by a type expression over the set of basic types B, a set of labels L and the constructors product (tuple), set, and bag,
- a set of operations defined in the ER algebra and limited to S, and
- a set of (static or dynamic) integrity constraints defined in the hierarchical predicate logic with the base predicate P_S.

The set of S-structured objects that fulfill the integrity constrains is called $\mathcal{S}^C$. A database schema $\mathcal{D} = (\mathcal{S}_1, ..., \mathcal{S}_n, \Sigma_G)$ is defined by a list $\mathcal{S}_1, ..., \mathcal{S}_n$ of different database types and a set of global integrity constraints. Based on a database schema views $\mathcal{V} = (V, O_V)$ can be defined using the HERM algebra to construct a parameterized expression V and a set of operations O_V applicable to V. Based on retrieval and modification operations in O_V we can derive an *input view* $I^{\mathcal{V}}$ and an output view $O^{\mathcal{V}}$.

A database component is a database scheme that has an import an an export interface for connecting it to other components by standardized interface techniques. It consists of input elements, output elements, and a database structuring. Components may be considered as input-output machines that are extended by the set of all states S^C with a set of corresponding input views $I^{\mathcal{V}}$ and a set of corresponding output views $O^{\mathcal{V}}$.
Input and output of components is based on channels. The structure of a channel K is described by the function $type : C \rightarrow \mathcal{V}$. Association of components is restricted to domain-compatible input/output schemata which are free of name conflicts. An output view O_1^V and an input view I_2^V are domain-compatible if $dom(type(O_1^V)) \subseteq dom(type(I_2^V))$.
The star schema is the main component schema used for construction. A star schema for a database type C_0 is defined by

- the full (HERM) schema $\mathcal{S} = (C_0, C_1, ..., C_n)$ covering all types on which C_0 has been defined,
- the subset of strong types $C_1, ..., C_k$ forming a set of keys $K_1, ..., K_s$ for C_0, i.e. $\bigcup_{i=1}^{S} K_i = \{C_1, ..., C_k\}$ and $K_i \rightarrow C_0, C_0 \rightarrow K_i$, for $1 \leq i \leq s$ and $card(C_0, C_i) = (1, n)$ for $1 \leq i \leq k$,
- the extension types $C_{k+1}, ..., C_m$ satisfying the (general) cardinality constraint $card(C_0, C_j) = (0, 1)$ for $(k + 1) \leq i \leq n$.

The extension types may form their own $(0,1)$ specialization tree (hierarchical inclusion set). The cardinality constraints for extension types are partial functional dependencies.
There are various variants for representation of a star schema, e.g. an entity type with specializations forming a specialization tree, or a relationship type with components $C_1, ..., C_k$, with attributes and a specialization tree.
A star component schema is usually characterized by a kernel entity type used for storing basic data and by a number of dimensions, usually based on subtypes of the kernel entity type, subtypes expressing additional properties, the life cycle, or categorization.

Star schemata may be extended to snowflake schemata. A snowflake schema is a

- star schema $\mathcal{S}$ on C_0 extended or changed by
  - variations $\mathcal{S}*$ of the star schema (with renaming)
  - with strong $1-n$-composition by association types $A_S^{S'}$ associating the star schema with another star schema $\mathcal{S}'$ either with full composition restricted by the cardinality constraint $card(A_S^{S'}, S) = (1,1)$ or with weak, referencing composition restricted by $card(A_S^{S'}, S) = (0,1)$,
- which structure is potentially C_0-acyclic

A schema $\mathcal{S}$ with a 'central' type C_0 is called potentially C_0-acyclic if all paths p, p' from the central type to any other type C_k are

- either entirely different on the database, i.e., the exclusion dependency $p[C_0, C_k]||p'[C_0, C_k]$ is valid in the schema
- or completely identical, i.e. the pairing inclusion constraints $p[C_0, C_k] \subseteq p'[C_0, C_k]$ and $p[C_0, C_k] \supseteq p'[C_0, C_k]$ are valid.

2.1.2 Conceptual Components

The framework for using database components can be adapted for content management systems. Database components defined in [Tha05] deal with the structural part of information. A similar framework is needed for the semantical part.
In a general discussion we assume a set $\mathcal{O}$ of objects and a set $\mathcal{R}$ of n-ary relations between objects. Conceptual components express the semantics of relating objects in a certain manner.
An object an be characterized by a set of properties that hold for that object as defined in section 1.4.3.
We can restrict connecting objects via relations by expressing integrity constraints formulated over the object's properties. If a relationship between objects is valid in terms of the application, the objects may expose new properties. Additionally,

properties for the whole compound can be derived. As shown in section 1.4 we can cluster sets of objects with similar properties in concepts.
The definition of a conceptual component is divided into two parts: the component's interface and the component's realization. The interface is given by the tuple $(\mathcal{N}, \mathfrak{O})$:

- $\mathcal{N}$ is a set of "named ports"'. A named port is characterized by a name n unique within the component and a concept $\mathfrak{P}^n$ describing the requirements that must be satisfied by another component to connect to the port n.
- The concept $\mathfrak{O}$ describes the properties of the component that are exposed to the environment. There are two different types of component properties:
  - properties that hold independently from connected components
  - properties that are derived from connected components

A conceptual component $C_1 = (\mathcal{N}_1, \mathfrak{O}_1)$ can connect to port n of component $C_2 = (\mathcal{N}_2, \mathfrak{O}_2)$, $n \in \mathcal{N}_2$ if $\mathfrak{O}_1 \sqsubseteq \mathfrak{P}_2^n$.

Conceptual components can be designed in three different ways: black-boxed, white-boxed or glass-boxed. A black-boxed component's realization defines the describing concept $\mathfrak{O}$ in a hidden way. A white-boxed realization is given by a formula φ in a language $\mathcal{L}$, e.g. first order predicate logic. A possible representation is the following:

- For every named port a variable is defined denoting a component that is connected to this port or a special value $undef$ if there is no such connection.
- There exist variables $x_1, ..., x_n$ bound by quantors that act as unnamed ports.
- For every property $p \in \mathcal{P}$ a function $p(x)$ is defined for port variables x and domain specific predicates such as equality. The function's value corresponds to the value of the property of the component that is connected to this component through port x.
- Based on these definitions first order predicate logic formulas can be build in the usual way.

Glass-boxed components expose the structure of their realization by aggregating other components to derive a certain concept. The realization of a glass-boxed component contains a set of component instances and a set of internal relations that map component instances to ports of other component instances, as well as to external ports (ports of the constructed component). A component instance is a component associated with a name that is unique within the upper component. Additional to the definition of the characterizing concept $\mathfrak{O}$ in the white-box case, concepts of aggregated component instances can be taken into account.

2.1.3 Relating Conceptual Components and Database Components

By now, we assume a component library. Each component is associated with a conceptual component describing the component's semantics and at least one database component definition describing the structural part. A conceptual component can be associated with more than one database component with the restriction that every associated database component meets the component's semantics but possibly under different circumstances or "soft requirements" (e.g. performance according to a defined performance measure). These "circumstances" can be clustered according to their "soft requirements" and can be treated as concepts. We call such concepts "architectures".
By using conceptual components the design process of database schema units extends:

1. A concept $\mathfrak{R}$ is created that represents the properties derived from the requirements analysis phase.

2. A component C_0^0 is constructed with $\mathfrak{R}$ as the characterizing concept.

3. The component is refined to a component in a glass-boxed way by introducing sub components that represent parts of the specification. This step is done iterativly. At any stage it can be checked whether the created unit satisfies the requirements specification ($\mathfrak{R} \sqsubseteq \mathfrak{O}_{C_i^j}$ and $\mathfrak{O}_{C_i^j} \sqsubseteq \mathfrak{R}$). Proofs of local properties can be made, too.

4. If a component C_i^j derived in step i can be represented by components C_L^k from the library ($\mathfrak{C}_i^j \sqsubseteq \mathfrak{C}_L^k$ and $\mathfrak{C}_L^k \sqsubseteq \mathfrak{C}_i^j$), then the designer might decide to place $\mathfrak{C}_L^k$ into the schema unit. The refinement process stops when every conceptual component within the component compound can be found within the component library.

5. In the next step the designer chooses an appropriate architecture and replaces the conceptual components by their structural counterparts.

6. The database components have to be connected on the structural level. Connections and ports on the conceptual level are replaced by channels. In the case of non-compatible views converter elements (see chapter 4) have to be introduced.

Using conceptual components allows a preselection of database components in large component libraries on an abstract level. Detailed features irrelevant for mounting components are hidden to the designer so the design becomes more understandable. Additionally, formal proofs of general system properties derived from the requirements analysis can be made on a smaller set of data which increases productivity.

2.1.4 Component Schema Construction

Composition of component based schemata usually follows certain design principles. The following methodologies are typical:

Constructor-Based Composition: Star and snowflake schemata may be composed by composition operators such as *product*, *nest*, *disjoint union*, *difference*, and *set* operators. These operators allow to construct any schema of interest since they are complete for sets. More natural approaches can be prefered, too: all constructors known for database schemata may be applied to schema construction.

Bulk Composition: Types used in schemata in a very similar way can be clustered together on the basis of a classification.

Architecture Composition: Categorization-based composition has been widely used for complex structuring. Architecture composition enables in associating through categorization and compartmentalization. This constructor is especially useful during modeling of distributed systems with local components and with local behavior. There are specific solutions for interface management, replication, encapsulation, and inheritance. The cell construction is the main constructor in component applications and in data warehouse applications. Therefore, composition by categorization is the main composition approach.

Lifespan Composition: Evolution of things in applications is an orthogonal dimension that must be represented in the schema from one side but which should not be mixed with constructors of the other side. We observe a number of lifespan compositions:

- *Evolution composition* records the stages of the life of things and is closely related to workflows.
- *Circulation composition* displays the phases in the lifespan of things.
- *Incremental composition* allows to record the development and the enhancement of objects as well as their aging.
- *Loop composition* supports changing and scaling to different perspectives of objects during their evolution.
- *Network composition* allows flexible treatment of objects during their evolution.

2.2 Application Scenario

Consider a library for graphical components ("widgets"[1]) produced by graphical designers that are enhanced by behavioral aspects. This section analyzes requirements for such a kind of content warehouse and discusses database schema modeling for this case from the perspecitive of a component based design.

2.2.1 Development of Graphical User Interfaces

Graphical User Interfaces (GUI) are often specified, designed, implemented, and tested in interdisciplinary, separated teams. These teams usually use special tools to achieve the development goals. To guarantee a seamless communication and data transfer between these teams a tool independent infrastructure is necessary.
The specification and implementation process is divided into several steps. In a first step general design guidelines are specified. According to these design guidelines the interface is designed in a pure graphical way. Later, these graphical prototypes are enhanced by adding functionality (the interface's behavior) to derive a logical prototype. In the last steps the logical prototype is transfered to a physical one by mapping the logical prototype to a desired framework.
Each step within this process needs specialized tools, e.g. graphic tools for designing the graphical prototype and tools for modeling state charts for the specification of the interface's behavior. The data used by single tools can be partially reused within other tools by transforming the data structures.
A designer should be able to reuse widgets based on a former design. Possibly, the designer is interested in little variations on the widget's representation.

2.2.2 Component Based Design

Looking at these requirements (a complete survey can be found in [KBF+05]) we identify the following facets from the perspective of a component based approach:

- The entities of central interest are "widgets". Widgets should be reuseable and versionable.
- Widgets are characterized by 5 dimensions:
  - *Graphics:* each widget has a graphical representation based on a composition of graphical primitives such as (filled) rectangles, polygons, (sp)lines, paths, text, etc.
  - *Behavior:* the behavior of a widget can be e.g. expressed by a state based approach such as state charts or simple FSMs, or event based, e.g. by event-condition-action (ECA) rules.

[1] A widget (window gadget) is a component visible on some kind of screen that has a certain behavior during the interaction with the user.

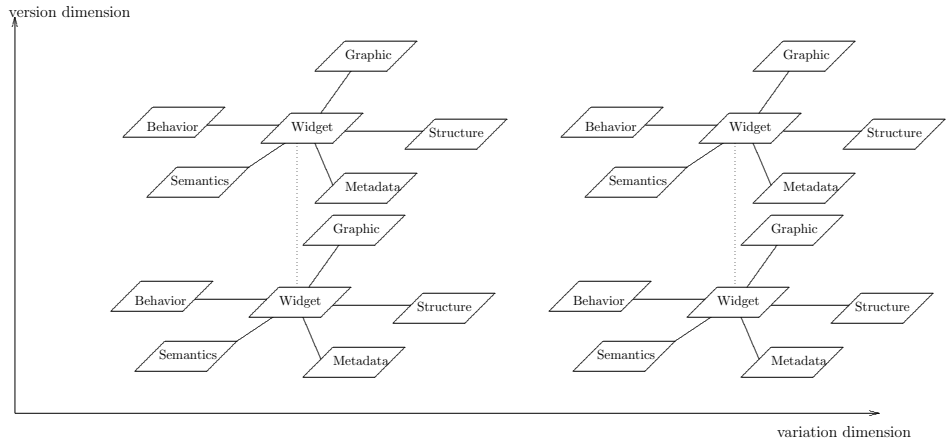


Figure 2.1: The Warehouse Structure for Graphical Components.

- *Structure:* a more complex widget can be composed of simpler ones.
- *Semantics:* a widget exposes properties and fits in certain application areas.
- *Metadata:* widgets are created and edited by certain authors, follow certain design rules, etc.

For every tool envolved in the development chain there is a view on this general definition of a "widget". Tools support own data structures that are not shared with other tools and not interpretable except for the tool. For that reason the definition of a widget should be "open": extensions should be allowed as long as the minimal requirements are satisfied.
For this reason the notion of "attractor types" is introduced: an attractor type is a kernel entity type that provides central properties such as identification. In terms of schema composition the attractor is used as the central type for a star schema. The attractor type defines sets O, P, F of types:

- For every type in O there must be an according dimension in the star schema (obligory types).
- For every type in P there can be an according dimension in the star schema (permitted types).
- For every type in F there must not be an according dimension in the star schema (forbidden types).

The condition for obligory types may be weakend according to a lifespan composition, e.g. according to the development chain in the example. Variations and versions can be derived by adding a new dimension *on the whole attractor compound*. Variations and versions differ in the way that there is a strict ordering assumed between versions. Figure 2.1 shows the general structure of the component construction.

3 Users and Rule Management

Every information system is embedded in an environment. External agents within this environment issue service requests to the information system. From the system's point of view it is not relevant whether an particular agent is a human being or an artifical one, e.g. another information system. Every agent is characterized by a set of needs. The Agent tries to fulfil these needs by triggering actions which are partially visible to other agents (and so to our system). Some of these actions may cause a service request to our information system. The agents's general plan for life is usually hidden to any other agent, but the information system can interpret the agent's actions and published properties on the base of its "knowledge" to extrapolate a model of the agent itself. These models can be used by the system to synchronize the needs of any interacting partner.

3.1 Agents, Actors, Roles, and Groups

We assume that any participating agent is known to and identifiable by the system. Usually, some kind of authentication and authorization facilitiy is used to accomplish this. If an agent is not able to authenticate itself or if the authentication process is not yet finished, the agent is usually identified as a *guest agent*. It is up to the system to allow guests or not. In the following we will assume a finite set $\mathcal{A} = \{a_1, ..., a_n\}$ of distinguishable agents. Further on, we will assume that two agents a_i, a_j are representations of different real world agents, even if a particular real world agent acts under two different identities.
Every agent a_i is associated with a partial property function $p_{a_i} : \mathcal{P} \rightarrow \mathcal{D}$ mapping properties $p_1, ..., p_m$ from a finite set $\mathcal{P}$ of properties to values from the corresponding domain $dom(p_i) \subseteq \mathcal{D}$. Additionally, agents are related to other agents to accomplish common goals. This is expressed by a set $\mathcal{R}$ of n-ary relations over $\mathcal{A}$, These properties and relations are the base for the system's imagination of the agent's plan for life. In the following discussion we assume $dom(p_1) = ... = dom(p_m) = \mathcal{D} = \{true, false\}$ for two constants $true$ and $false$. If any property p_i is not a boolean one it can be replaced by a set $\{p_i^1, ...p_i^k\}$ of boolean properties p_i^l using an appropriate scaling function or by introducing predicates over the domains such as the equality relation. A discussion of scaling multivalued properties can be found in [GW98].

Every agent is an individual with its own properties. From the point of view of the system the individual status is not very important. The system clusters agents with an estimated equivalent plan for life and treats them equally. That's why we

introduce the notion of *actors*: an actor $\mathfrak{A}$ is a concept (ϱ, G) where ϱ is a formula (of a language $\mathcal{L}^{\mathfrak{A}}$) called *role*, $G \subseteq \mathcal{A}$ is a set of agents called *group* with $\mathcal{E}(\varrho) = G$ and $\mathcal{I}(G) = \varrho$. Depending on the system's capabilities an appropriate language $\mathcal{L}$ has to be defined to express roles. First order predicate logic is usually sufficient. Like any other concept actors can be defined either intensionally by supplying the role formula ϱ or extensionally by listing the actors's group. In the following, we use the predicate $A(x)$ which is true for all objects x in the extension of $\mathfrak{A}$: $A(x) \Leftrightarrow (x \in \mathcal{E}(\mathfrak{A}))$.

3.2 Task Obligations and Access Rights

An agent interacts with the system by requesting operations on the system's objects. We define system objects in a general way. A system object can be any component of the system an actor can operate with. System objects may be related to other system objects, e.g. `part-of` relationships. Operations may subsume other operations. Like agents can be conceptually treated as actors, objects and operations can be treated by appropriate concepts. We will denote the set of objects with $\mathcal{O}$ and the set of operations with $\mathcal{M}$. $\mathcal{C}$ is the set of all predicates involving actors, object concepts, and operation concepts as well as combinations of them (like $isPart(o_1, o_2)$ or $worksOn(a, o)$).
Given a triple (a, o, m) of an agent a, an object o and an operation m, the central question for the system are

1. "Is it an obligation for agent a to execute operation o on object o?"

2. "Is the agent a permitted to execute operation m on object o?"

The general assumption is that the system will allow any operation that has to be executed by an agent, but it will not allow any operation that is not necessary in any situation.
Permission to execute a certain operation is granted temporary based on the relationships between agents and objects. We can define a universe of executions containing all triples (a, o, m) that are considered to be possible:

$\mathcal{U}^P = \{(a, o, m) | \textit{there is a possible situation, where } a \textit{ executes } m \textit{ on } o\}$

Dually, we can define the universe of all forbidden executions:

$\mathcal{U}^F = \{(a, o, m) | \textit{there will never be a valid situation, where } a \textit{ executes } m \textit{ on } o\}$

Usually, one universe is defined and the other one is derived by applying one of the rules

- Everything is forbidden that is not permitted.

- Everything if permitted that is not forbidden.

Obligations and permissions can be defined either statically or dynamically. Static permission definitions do not consider the actual execution context to determine whether access has to be granted or not. Dynamic behavior is usually derived by applying the operations `grant` and `revoke` to update the current set of obligations and permissions over time.

A simple static permission definition can be seen as a matrix associating actors and objects to granted permissions on operations. For example, consider a document server serving documents for editing and publishing. Registered users have to authenticate, so the user's relevant properties are known. We assume the following operations on documents:

read - show the content of the document
update - change the content of the document
branch - create a new document that is initialized with the current content
review - check whether the document is ready for publication or not
publish - make the document accessible by the world

The following matrix shows a sample configuration for access rights: editors are allowed to read documents and to create a new copies, the owner of a document is allowed to update the document and to publish a reviewed document, a reviewer is allowed to check the content of the document and everybody is allowed to read a published document. We assume that the underlying infrastructure supplies the predicates visible in the first column. To keep the matrix compact we omit type checking and denote actors with a and documents (objects) with o.

	read	*update*	*branch*	*review*	*publish*
$Editor(a)$	×		×		
$Owner(a, o)$	×	×			
$Reviewer(a)$	×			×	
$Owner(a, o) \wedge reviewed(o)$	×				×
$published(o)$	×				

A × marker within the matrix denotes that the corresponding operation is allowed. We can associate different semantics with this matrix.

3.2.1 Prerequisites for Execution

We define a predicate $exec(a, o, m)$ which denotes "actor a executes operation m on object o". We derive a formula in first order predicate logic from the matrix: for every operation m_i we consider the prerequisites that are necessary to execute the operation:

$$
\begin{array}{rcll}
F_1 = (exec(a,o,read) & \Rightarrow & Editor(a) \vee Owner(a,o) \vee Reviewer(a) \vee & \\
 & & (Owner(a,o) \wedge reviewed(o)) \vee published(o)) & \wedge \\
(exec(a,o,update) & \Rightarrow & Owner(a,o)) & \wedge \\
(exec(a,o,branch) & \Rightarrow & Editor(a)) & \wedge \\
(exec(a,o,review) & \Rightarrow & Reviewer(a)) & \wedge \\
(exec(a,o,publish) & \Rightarrow & Owner(a,o) \wedge reviewed(o)) &
\end{array}
$$

Given a requested operation execution $(a', o', m') \in \mathcal{U}^P$ and an interpretation $\mathcal{I}$ with $(a', o', m') \in \mathcal{E}(exec)$ access is granted if there exists a valuation σ that assigns a to the value of a', o to the value of o',and m to the value of m' such that $\models_{\mathcal{I},\sigma} F_1$.

3.2.2 Deriving Permissions for Execution

Expressing prerequisites for executions is efficient for checking whether operations can be executed or not: given an execution request (a', o', m') you search for the right implication to check. There is no need to completely construct *exec*'s extension or the complete valuation σ. Unfortunately, all prerequisites of every operation have to be completely unfolded, dependenices between operations are not considered. Additionally, it is not possible to derive all permissions for an agent efficiently.
If we consider permissions instead of prerequisites we are able to handle these disadvantages. We define a predicate $grant(a, o, m)$ which denotes "agent a is permitted to execute operation m on object o". Now we can write down the rows of the given matrix. Because the rows are no access restriction we have to add the execution prerequisites:

$$
\begin{array}{rcll}
F_2 = (Editor(a) & \Rightarrow & grant(a,o,read) \wedge grant(a,o,branch)) & \wedge \\
(Owner(a,o) & \Rightarrow & grant(a,o,read) \wedge grant(a,o,update)) & \wedge \\
(Reviewer(a) & \Rightarrow & grant(a,o,read) \wedge grant(a,o,review)) & \wedge \\
(Owner(a) \wedge reviewed(o) & \Rightarrow & grant(a,o,read) \wedge grant(a,o,publish) & \wedge \\
(published(o) & \Rightarrow & grant(a,o,read) & \wedge \\
(grant(a,o,read) & \Rightarrow & Editor(a) \vee Owner(a,o) \vee Reviewer(a) \vee & \\
 & & (Owner(a,o) \wedge reviewed(o)) \vee published(o)) & \wedge \\
(grant(a,o,update) & \Rightarrow & Owner(a,o)) & \wedge \\
(grant(a,o,branch) & \Rightarrow & Editor(a)) & \wedge \\
(grant(a,o,review) & \Rightarrow & Reviewer(a)) & \wedge \\
(grant(a,o,publish) & \Rightarrow & Owner(a,o) \wedge reviewed(o)) &
\end{array}
$$

Given a requested operation execution $(a', o', m') \in \mathcal{U}^P$ and an interpretation $\mathcal{I}$ with $(a', o', m') \in \mathcal{E}(grant)$ access is granted if there exists a valuation σ that assigns a to the value of a', o to the value of o',and m to the value of m' such that $\models_{\mathcal{I},\sigma} F_2$. Additionally, we can conclude all permissions for an actor or an object.

Relationships between operations can also be taken into account. Assume, that it is necessary to read a document before you can create a new branch, review or publish it. The permission matrix simplifies:

	read	*update*	*branch*	*review*	*publish*
$Editor(a)$			$\times$		
$Owner(a, o)$	$\times$	$\times$			
$Reviewer(a)$				$\times$	
$Owner(a, o) \wedge reviewed(o)$					$\times$
$published(o)$	$\times$				

We can express these dependencies by adding new implications to our formular:

$$\begin{aligned} F_3 = F_2 \wedge \quad & (grant(a, o, branch) \Rightarrow grant(a, o, read)) \wedge \\ & (grant(a, o, review) \Rightarrow grant(a, o, read)) \wedge \\ & (grant(a, o, publish) \Rightarrow grant(a, o, read)) \end{aligned}$$

In a similar way static dependencies between actors or objects (e.g. part-of relationships) can be handled.

3.2.3 Permitted and Forbidden Operations

To calculate permissions and restrictions efficiently it is important to keep the prerequisites as simple as possible. Conjunctions or disjunctions of simple predicates can be evaluated very quickly while expressions with quantors can be very costly. In many cases it is more elegant to explicitly separate permitted and forbidden operations. Therefore, two predicates *grant* and *revoke* are defined.
Given such a formula F, a requested operation execution $(a', o', m') \in \mathcal{U}^P$ and an interpretation $\mathcal{I}$ with $(a', o', m') \in \mathcal{E}(grant), (a', o', m') \notin \mathcal{E}(revoke)$ access is granted if there exists a valuation σ which assigns a to the value of a', o to the value of o',and m to the value of m' such that $\models_{\mathcal{I},\sigma} F$. Depending on the system's requirements weaker conditions like $(a', o', m') \in \mathcal{E}(grant)$ or $(a', o', m') \notin \mathcal{E}(revoke)$ may be appropriate, too.
Typically, calculation of permitted and forbidden operations is separated into two steps: first, it is determined whether the operation is permitted or not. If the operation is permitted, a second calculation against another access specification is made to determine whether the operation is forbidden or not. This procedure is efficient if you have a lot of permissions and only specific restrictions. You can swap the calculations in the case where access is usually forbidden but allowed under certain conditions. Access is granted if the execution triple passes both tests.

3.2.4 Logic of Actual Obligation: Dynamic Permissions

Dynamic definitions grant or revoke access rights depending on "circumstances" based on the system's state and history. Dynamic permission definitions subsume

static ones because every static permission applies under any "circumstance".
We follow the logic of actual obligation (**LAO**) to express executions, obligations, and permissions. **LAO** combines temporal and deontic aspects under consideration of actions.

Logic of Actual Obligation

This section shortly introduces **LAO** . For further information see [Voo89].
The basis of **LAO** is a set of possible worlds $u, v, w, ...$ with a partial order $<$. $<$ is assumed to be transitive, irreflexive, tree-like, and serial where $v < w$ means that w is a possible future world of v. At each world actions can be done:

- $a, b, c, ...$ are elementary actions
- U and $\varnothing$ are special actions: U is the universal action, $\varnothing$ is the empty action.
- if a and b are actions, then the following constructs are actions, too:
  - $\neg a$ is the action "not a", denoting that a will never be done
  - $/a$ is the action $a - complement$, denoting "not doing a"
  - $a; b$ is the sequential composition of a and b (first a is done, then b is done)
  - $a \parallel b$ is the parallel execution (with the restriction that the result of doing a does not conflict with the result of doing b)
  - $a + b$ is the choice (either a, or b, or both)
- Nothing else is an action.

Constructs like $a \otimes b$ (either a or b, but not both), $\parallel \{a, b, c, ...\}$, $+\{a, b, c, ...\}$, or $\otimes\{a, b, c, ...\}$ can be defined.
In every world v there is a pre-ordering $\leq_v$ between actions with the properties $a \leq_v b \wedge b \leq_v c \Rightarrow a \leq_v c$ and $a \leq_v c$ and $b \leq_v c \Rightarrow a + b \leq_v c$. $a \leq_v b$ has the intended meaning that "action b is at least as good as a".
LAO defines assertions as follows:

- Every propositional atom is an assertion.
- If φ, ψ are assertions, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$, $\Box\varphi$, $\Diamond\varphi$, and $\hookleftarrow \varphi$ are assertions.
- If a and b are actions, then $a \rightarrow b$, $a \equiv b$,$a \leq b$, $\Box a$, $\Diamond a$, $D(a)$, $O(a)$, $O'(a)$, $O^*(a)$, $O * (D(a))$, $O^S(a)$, $P^S(a)$ are assertions.
- If i is a positive integer, a is an action, and φ is an assertion, then $\Box^i\varphi$, $\hookleftarrow^i \varphi$, $D^i(a)$ are assertions.

- If φ is an assertion and a is an action, then $[a]\varphi$ is an assertion.
- If $\varphi(\boldsymbol{a})$ is an assertion with a free variable $\boldsymbol{a}$ (denoting actions), then $(\forall \boldsymbol{a})(\varphi(\boldsymbol{a}))$ and $(\exists \boldsymbol{a})(\varphi(\boldsymbol{a}))$ are assertions.

The intended meanings of some assertions are

- $a \to b$: $(\forall w)(v <_a w \Rightarrow v <_b w))$
- $a \equiv b$: $a \to b$ and $b \to a$
- $\hookleftarrow \varphi$: φ has been the case somewhere in the past.
- $\hookleftarrow^1 \varphi$: φ was the case in the possible world immediate before this one. $\hookleftarrow^{i+1} \varphi \Leftrightarrow \hookleftarrow^1 \hookleftarrow^i \varphi$
- $D(a)$: a has been done.
- $\Box^1 \varphi$: In all immediate successors of this world φ is the case. $\Box^{i+1} \varphi \Leftrightarrow \Box^1 \Box^i \varphi$
- $[a]\varphi$ if a will be done, then φ will be the case
- $O'(a)$ denotes an acute duty: $O(a)$ requires no immediate action. If not doing a under obligation $O(a)$ implies doing $\neg a$ then a must be done immediately: $O'(a) :\Leftrightarrow (\exists b)(O(b) \wedge [/b]D(\neg b) \wedge b \to a)$
- $O^*(a)$ denotes an actual obligation: $O^*(a) :\Leftrightarrow (\exists b)(O'(b) \wedge b \leq a \wedge (\forall c)(c \geq b \Rightarrow c \to a))$
- $O^S(a)$ and $P^S(a)$ are strong obligations and strong permissions: $O^S(a) :\Leftrightarrow O^*(a) \wedge (\forall b)(O^*(b) \Rightarrow a \to b)$ $P^S(a) :\Leftrightarrow \neg O^*(/a)$

A **LAO** -model $\mathbb{M}$ is a tuple $\langle K, <, E, N, \Vdash \rangle$ where K is a set of possible worlds, $<$ is a transitive irreflexive tree-like serial partial ordering on K, E is the set of atomic actions. An action is interpreted as a relation over $K \times K$. N is a set of formulas of the form $\varphi \Rightarrow O(a)$ for an action a and an assertion φ that does not contain $\leq$ or the O operator. N has to satisfy the condition, that there is no obligation for doing an action that prevents another obligation.
$\Vdash$ is a forcing relation:

- $v \Vdash a \leq b :\Leftrightarrow a \leq_v b$
- $v \Vdash a \to b :\Leftrightarrow (\forall w)(v <_a w \Rightarrow v <_b w)$
- $v \Vdash \Box\varphi :\Leftrightarrow (\forall w > v)(w \Vdash \varphi)$
- $v \Vdash \Box^1\varphi :\Leftrightarrow$ for all direct successors w: $w \Vdash \varphi$

- $v \Vdash \hookleftarrow \varphi :\Leftrightarrow (\exists w < v)(w \Vdash \varphi)$
- $v \Vdash \hookleftarrow^1 \varphi :\Leftrightarrow$ there exists a direct predecessor w: $w \Vdash \varphi$
- $v \Vdash [a]\varphi :\Leftrightarrow (\forall w >_a v)(w \Vdash \varphi)$
- $v \Vdash D(a) :\Leftrightarrow$ there exists a $w <_a v$
- $v \Vdash D^1(a) :\Leftrightarrow$there exists a direct predecessor$w <_a v$
- $v \Vdash (\forall \boldsymbol{a})(\varphi(\boldsymbol{a})) :\Leftrightarrow$ for all actions b $v \Vdash \varphi(\boldsymbol{a}|b)$
- $v \Vdash (\exists \boldsymbol{a})(\varphi(\boldsymbol{a})) :\Leftrightarrow$ there exists an action b so that $v \Vdash \varphi(\boldsymbol{a}|b)$
- $v \Vdash O(a) :\Leftrightarrow v \Vdash \neg\Box\neg D(a)$ and for some φ: $v \Vdash \varphi$ and $\varphi \Rightarrow O(a) \in N$

Applying LAO to Access Permissions

In the following we consider runs of the system. A run is a sequence of executed operations[1] from $\mathcal{U}^P$ and can be seen as an isolated transaction. We assume, that there are no causal dependencies between different runs within the system except that they share the same underlying infrastructure.
The set of possible atomic actions is given by $\mathcal{U}^P$. Given a moment in time t the tuple (t, r_t) with a finite run r_t can be seen as a possible world in terms of **LAO**. The relation $v < w$ between two runs is defined as follows: $t_v < t_w$ and $r_{t_w} = append(r_{t_v}, a)$ (with an executed operation a) or $r_{t_w} = r_{t_v}$.
As usual, we define the permission operator $P(a) = \neg O(/a)$ (it is permitted to execute a if there is no obligation for not executing a.)
The system of norms N contains the actual access rights definitions in the form $\varphi \Rightarrow P(a)$ or $\varphi \Rightarrow O(a)$. For our document server the following norms apply:

$$
\begin{aligned}
&Editor(a) \Rightarrow P(a, o, read) \wedge P(a, o, branch)\\
&Owner(a, o) \Rightarrow P(a, o, read) \wedge P(a, o, update)\\
&Reviewer(a) \Rightarrow P(a, o, read) \wedge P(a, o, review)\\
&Owner(a, o) \wedge released(o) \Rightarrow P(a, o, read) \wedge P(a, o, publish)\\
&published(o) \Rightarrow P(a, o, read)
\end{aligned}
$$

Given a model $\mathbb{M} = \langle K, <, \mathcal{U}^P, N, \Vdash\rangle$, a requested action (a, o, m), and a run r representing the current partial run there are two possibilities to define access permissions:

- **Task centered:** access is granted if there is an obligation to do some action ($\mathbb{M}, r \models O(a, o, m)$). This is sufficient for closed systems defining strict workflows.

[1] In the case of *real* parallelism it is a sequence of sets of operations.

- **Filtering:** access is granted if there is an obligation or a permission to do some action ($\mathbb{M}, r \models P(a, o, m)$).

The following example shows the ability to express access permissions depending on actions that took place in the past. Assume the following scenario from a library:

Registered users (predicate user) are permitted to borrow available books (predicates book and available)[2] *for 30 days. After this period the user has to bring back the book. The deadline can be extended twice.*

We identify three operations: *borrow*, *bringBack*, and *extend*. The partial preordering $\leq_v$ between actions is defined as $(a, o, extend) \leq_v (a, o, bringBack)$ for every world v and arbitrary values a and o. We construct N by introducing the following norms (assuming a timing resolution[3] of 1 day.):

- Registered users are permitted to borrow books:

$$user(a) \wedge book(o) \wedge available(o) \Rightarrow P(a, o, borrow)$$

- After 30 days the user has to bring back the book or has to extend the deadline:

$$D^i(a, o, borrow) \Rightarrow O^{i+30}((a, o, bringBack) + (a, o, extend))$$

- The deadline can be extended twice:

$$\hookleftarrow^i (\hookleftarrow^1 (D(a, o, extend)) \wedge D^1(a, o, extend)) \Rightarrow O^{i+30}(D(a, o, bringBack))$$

3.3 Implementing Access Control Lists

Checking access is the most important operation in user management because it has to take place every time an execution request occurs. In this section we will discuss how the specifications from the last sections can be supported by a content management system using techniques from query processing in databases.

3.3.1 Static Permissions

A static permission definition contains implications that map associations of actors and objects to granted operations. We assume that the number of possible operations

[2]Please note: we defined runs as isolated transactions without any causal dependencies between runs. To express "another user borrowed the book somewhere in the past" needs to integrate all user transaction in one big system transaction. This makes expressing the norms more difficult and counterintuitive. These problem can be faced by introducing hubs between runs: selected properties of the set of runs are exposed to the runs in terms of predicates.

[3]Timing is usually unspecified in **LAO**. In this example we assume a one-to-one mapping between a logical timing unit and a real-time period of 1 day

is rather small, while the number of actors and (especially) objects can be large. An implication as defined above follows the structure

$$grant(a, o, m) \Rightarrow \varphi_1 \vee ... \vee \varphi_n$$

Operations depending on others (defined by formulas $grant(a, o, m_1) \Rightarrow grant(a, o, m_2)$) can be unfolded by adding the prerequisites for executing m_1 to the prerequisites for executing m_2.
Following the lazy evaluation approach the formula $\varphi_1 \vee ...\varphi_n$ evaluates to true iff we find a φ_i that evaluates to true.
A formula φ_i can be of some structure. In many cases, it is a conjunction or disjunction of formulas $\varphi_{i,j}$, but we do not depend on it. A conjunction φ_i of formulas is evaluated to true iff we do not find any formula $\varphi_{i,j}$ that evaluates to false.
To reduce evaluation time we reorder the formulas φ_i such that we can expect to skip as many formulas φ_i as possible. Each formula φ_i can be associated with a proximity P_{φ_i} of evaluating the formula φ_i to true. During evaluation the system has to pay certain costs, e.g. hard disk access or CPU time. In the following discussion these costs will be denoted by $C^t_{\varphi_i}$ in the case of successful evaluation (to true) and by $C^f_{\varphi_i}$ otherwise.
With $\langle \varphi_1, ..., \varphi_n \rangle$ we denote a list of formulas in evaluation order. φ_1 is evaluated first. If φ_1 does not evaluate to true, we evaluate φ_2 and so on. We can calculate the expectation value of the total costs of evaluating φ:

$$\begin{aligned} C^t_{\varphi_1,...,\varphi_n} &= P_{\varphi_1} \cdot C^t_{\varphi_1} + (1 - P_{\varphi_1}) \cdot (C^f_{\varphi_1} + C^t_{\varphi_2,...,\varphi_n}) \\ C^f_{\varphi_1,...,\varphi_n} &= \sum_{i=1}^{n} C^f_{\varphi_i} \end{aligned}$$

This cost estimation does not consider subsumption of formulas φ_i and φ_j. A similar discussion can be made for conjunctions and other language constructs.
There are $n!$ possibilities to construct evaluation orders for n formulas. Because it is not possible to enumerate these possibilities efficiently (and because the proximities and costs are only estimations, too) we have to prune the decision space.
We assume that it is possible to categorize formulas in the following way:

1. formulas with a high proximity to evaluate to true and with low costs
2. formulas with a high proximity to evaluate to true but with high costs
3. formulas with a low proximity to evaluate to true and with low costs
4. formulas with a low proximity to evaluate to true and with high costs

Formulas of the first category should be evaluated first while formulas from the last category should be evaluated at the end. Within a category we order the formulas with increasing costs.

Proximities are estimated by relative frequencies $\frac{|\varphi|}{|\mathcal{U}|}$ where $|\varphi|$ is the (estimated) size of φ's extension and $|\mathcal{U}|$ is the size of the universe[4]. Costs and proximities are calculated following the syntactical rules:

- If φ is an extensional predicate (a predicate backed by e.g. a database relation), C^t_φ and C^f_φ depend on the physical organisation of the predicate's extension within the database, e.g. if the extension is organized (or indexed) as a dynamic hashtable 2 disk reads are necessary while a heap-like organisation is more expensive (see [HR01]).
- $C_{\neg\varphi} = C_\varphi$, $P_{\neg\varphi} = 1 - P_\varphi$
- $C_{\varphi\wedge\psi}$ and $C_{\varphi\vee\psi}$ as defined above,
  $P_{\varphi\wedge\psi} < min(P_\varphi, P_\psi)$
  $P_{\varphi\vee\psi} > max(P_\varphi, P_\psi)$
- $C_{(\exists x)(\varphi)}$: a pessimistic estimation: $|\mathcal{U}| \cdot C_\varphi$
- $C_{(\forall x)(\varphi)}$: a pessimistic estimation: $|\mathcal{U}| \cdot C_\varphi$
- If there is some cycle (recursive defined predicates) we assume maximum costs and lowest proximity.

These values are estimations at the time the access rights were defined. During the system's run the execution plan can be adapted to the current behavior. Every condition φ_i is associated with a number ("hits") that counts the evaluations where φ_i triggered the decision (where φ_i was the first condition in the chain that was evaulated to true.) After a certain period of time the evaluation plan is reinvestigated: if φ_{i+1} has more hits than φ_i (plus some threshold value to prevent trashing) both conditions are swapped. Better approximations of the predicate's extensions can be taken into account, too. The strategy can be improved by applying materialization: if (a, o, m) was once evaluated to true, the system can remember this decision as long as the underlying predicates are not updated.

3.3.2 Dynamic Permisssions

There is an axiomatization of **LAO** so reasoning is supported. For access checking this is not sufficient. There are two possible strategies for handling dynamic permissions:

- There exists a mapping between a state transition description (e.g. a workflow language) and **LAO** formulas. Access checking can be made by evaluating the workflow while **LAO** is used to reason about general system properties.
- Allowed language constructs are limited syntactically.

[4]We can assume a finite universe — which is potentially evolving

For many cases permission and obligation definition can be written in the following form:

$$\varphi_0 \wedge \hookleftarrow^{f_1(i_1)} (\varphi_1 \wedge \hookleftarrow^{f_2(i_2)} (\varphi_2 \wedge \hookleftarrow^{f_3(i_3)} (\ldots \wedge \hookleftarrow^{f_n(i_n)} (\varphi_n)))) \Rightarrow \begin{cases} P^{f(i_1,\ldots,i_n)}(a) \\ O^{f(i_1,\ldots,i_n)}(a) \end{cases}$$

φ_i are assertions containing first order predicate logic formulas and D clauses. The intensional meaning is: if certain actions were done in the past, there arises an obligation/permission in the future. Detecting relevant situations may be still very cost intensive, but if a classification for situations exists, an online string matching algorithm (see e.g. [Koz00]) can be used to detect relevant situation sequences. If such a sequence occurs, obligations and permisssions are queued and can be used for access determination.

4 Data Exchange

Data stored in a content warehouse is queried by external applications or other content management systems. These systems have their own specifications of data structuring and semantics. This chapter discusses architectures that cope with data integration issues first and derives a converter architecture for connecting content warehouses to external content.

4.1 Data Integration in General

Database integration is currently solved only for the case of simple structures. Semantics is mainly neglected. It is known but often neglected that database integration cannot be automated. System integration is far more difficult. Both integrations can only be performed if a number of assumptions can be made for the integrated system. Instead of integrating systems entirely cooperation or collaboration of systems can be developed and used.
The difficulty of database integration is caused by

- the heterogeneity of data both at the intensional and extensional level,
- limitations to access the source data,
- the decision what data should be materialized and what should be left to local databases,
- data extraction, cleansing and reconciliation within the database set,
- strategies for data modification processing,
- strategies for quality management of querying, especially statements on whether the data in the query answer is complete and sound,
- automatic transformation of queries posted to the database set, and
- expressiveness of modeling languages aiming at representing the local databases and the integrated databases.

A main problem to be solved in designing such integrated systems lies in *information integration*, i.e., the activity by which different input information sources are merged into a global system describing the whole information set available for query and functionality purposes. Abstraction amounts to clustering types belonging to

the schema [BM99] into homogeneous subsets and producing an abstracted schema obtained by substituting each subset S with a single abstract type T^S.
Already structural integration (e.g. [CP98; BR01; ACM97; MCGF99; CSS99]) may become a nightmare. The designer has to clearly understand the semantics of involved database types. In such system re-engineering problems, the design emphasis is on integration of pre-existing information components. A key problem is deriving associations holding among types in the pre-existing schemas. Most of research has been carried out to solve the problem of detection and treatment of *interscheme properties* that relate types belonging to different schemas. The integration of structures and functions [TH02] is far more difficult.
Structural integration problems [Tha00] such as *structural mismatches* (key differences, abstraction grain, attribute domain, temporal basis, missing parts) *semantic mismatches* (scope difference, value semantics, domain semantics), *operational mismatches*, and *application domain mismatches* may be either treated by full integration, integration by merging, or integration by generalization. In the literature, many *"manual" methods* [Tha00] for deriving interschema properties have been proposed. A major limit of manual methods relies in the difficulty of carrying them out to large applications since, in such contexts, it is needed to face integration problems often involving hundreds of types.
Since an automatic support to integration of systems cannot be developed, *semi-automatic methods and tools* have been developed or proposed to face the difficulties. Systems such as Autoplex, automatch, Clio, COMA, Cupid, Delta, DIKE, EJX, GLUE, LSD, MOMIS, ARTEMIS, SemInt, SKAT, Similarity Flooding (SF), and TranScm mainly have emerged from specific applications. A very few approaches (Clio, COMA, Cupid, and SF) try to address the schema matching problem in a generic way. All of them are, however, only treating simple structural concepts and none of them treats functionality.
Integration of several information resources requires, however, knowledge describing their contents in a logical formalism and using the same vocabulary. This provides shared access to multiple information sources and preserves at the same time the autonomy of each source. This approach is known as the *mediator* approach [Wie95; CDSS98; PV99; LG99]. Mediators play the role of an interface between the user and the sources and between the sources giving the illusion of querying a central and homogeneous system.
Potential database integration depends on early modeling assumptions and is thus dependent on a number of *implicit assumptions* made during the development process:

- Database development is ruled by a number of implicit points of view. Depending on what has been the main target and scope, basic structures and domains are chosen.

- Development of database structuring is often ruled by the intentions for the utilization of the database. These intentions are based on main functionality of

the database. Normalization and later denormalization shows that functional requirements may be conflicting.

- Discretization of data or conversion of continuous data to discrete data will lead to different behavior and different query facilities of databases. Discretization may be based on time, space and other abstractions which may vary depending on the point of view the specific application is considered at the time of the development.

- Database developers make their assumption on the name space to be used. Name spaces depend on the concepts used in the application area.

- The chosen modeling language imposes a number of restrictions to structuring of the database. Some of these restrictions are unnatural, do not apply to the implementation platform, and lead to introduction of artificial types that do not have a meaning in the application area.

- The scope of data representation is often concentrated on the scope of the user at the business user level. This restriction takes to representation of macro-data which are comprehensions of micro-data that must have been used in the database.

- Data abstractions are often used instead of basic data. Since abstractions ease querying, systems are faster. At the same time, modification might be very complex. If abstractions are used then the application has to be remodeled to basic data structuring supported by view processing for computation of data abstractions.

- Optimization of structuring to performance and tuning uses normalization techniques. Since the same set of constraints might take to different normalized structures, optimization decisions must be made explicit.

One kind of difficulties of the database integration problem is caused by the development culture which does not force these implicit assumptions to be accessible.
Database integration has been discussed over a long period of time. A negative result that is often neglected in research and applications is the following ([Con86]):
The problem whether databases can be integrated is undecidable.
However, it is an observation often made in applications that these application databases can be integrated.
Databases to be integrated can be considered as views of the integrated database. Therefore, the integrated database should support the entire application. In the past, three approaches [CGL$^+$98; Tha00] have been worked out to treat integrated databases:

- *Global-as-view integration (GAV):* The integrated database is virtual. In reality, the local databases are still running on their own. There are no common

functions or queries. GAV supports a *client-driven integration* and bottom-up development and extension of local source systems. The GAV approach reduces query processing to view processing.

- *Local-as-view integration (LAV):* [Len02] The database integration allows us to build a data warehouse containing all data of the local application. The data of the local application corresponds to virtual or materialized views of the global database. Some change of the local data is harmonized with the global data if the change is going to be supported. LAV supports *source-driven integration* of applications and top-down design of applications by incremental addition of new sources. The global integration of all local databases supports consistency of all data and rejects wrong modifications of the database in a very early stage. The integration effort is, however, rather high. LAV often forces a reconsideration of the local schema. Some of the local applications must be redeveloped and reimplemented.

- *View cooperation:* [Tha00] Database cooperation is supported by exploiting the import/export facilities of the local databases. Each of the local database systems provides a number of views to the other databases. These views are either export views or import views of the collaborating databases. The schema of an importing view of the importing system contains the schema of an exporting view of the exporting system. View cooperation combines the local-as-view and the global-as-view approaches while maintaining their advantages. The mapping of the databases is similar to LAV mappings.

  The view cooperation approach is at the same time the most general approach. We may immediately derive the following corollary: *Local-as-view integration and global-as-view integration can be expressed through view cooperation expressions.*

Often full integration is not the aim. The aim is to achieve consistency. In this case the views should be (pair-wise) consistent via some translation mechanism: databases cooperate. This *database cooperation* mechanism is based on the construction of functions mapping parts of the view instances on parts of the other view instances. The next generalization step is to build the interface mechanism as a whole.

4.1.1 Database Cooperation

The integration methods discussed above use inheritance of IsA-relationship types: all attributes and operations of a metaclass are propagated to their subclasses unless overridden explicitly by a subclass. Explicit definition of the cooperation functions is more general. We say that one view A *dominates* the view B if a set of formulas exists such that the types of the view A can be embedded into B . Thus the view integration problem determines whether a minimal schema exists for a collection of views such that the schema dominates each of the views.

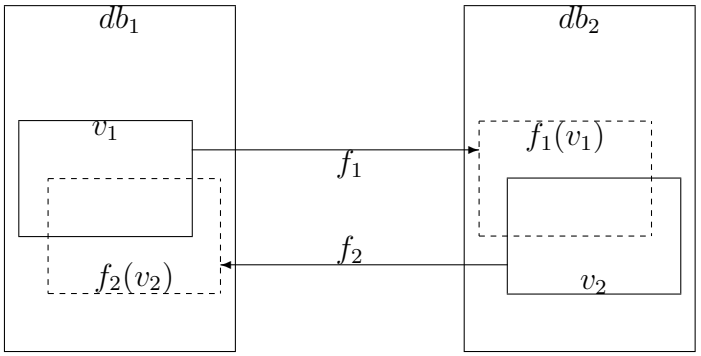


Figure 4.1: View Cooperation in Databases

We say that the views A, B *cooperate* via the (partial) functions f_A, f_B defined on $SAT(A), SAT(B)$
$f_A : SAT(A) \dashrightarrow SAT(B)$
$f_B : SAT(B) \dashrightarrow SAT(A)$
if for each $v_A \in SAT(A)$, $v_B \in SAT(B)$ the functions $f_A(v_A)$, $f_B(f_A(v_A))$, $f_B(v_B)$, $f_A(f_B(v_B))$ are defined and $f_B(f_A(v_A)) = v_A$, $f_A(f_B(v_B)) = v_B$.
The functions for view cooperation can be composed of functions in different parts of the view. Generally speaking, views cannot be completely mapped onto each other. Therefore, to decide whether two views cooperate we need to complete the following tasks:

1. 1. Find parts of the two views which are candidates for cooperation.
2. 2. For these candidates find the corresponding cooperation functions.
3. 3. Compose the view cooperation functions.

In order to establish whether parts of a view cooperate with other parts we use semantic information about the views.
The question whether views can be integrated or can cooperate can be answered if semantics of the views are well-defined. Integrity constraints can be used for this purpose. If one of the subset relationships is valid then the corresponding types can be embedded into their supertypes. This approach can be extended to view cooperation as displayed in Figure 4.1.
Assume schemata S_1, S_2 and selectors sel_1, sel_2 defined on S_1, S_2. The views V_1, V_2 can be defined by the given selectors. Furthermore, take two (S_1, S_2, sel_1, sel_2) functions
$f_1 : SAT(V_1) \dashrightarrow SAT(V_2)$
$f_2 : SAT(V_2) \dashrightarrow SAT(V_1)$.
We notice that $SAT(V_i) = sel_i(SAT(S_i))$ for $i = 1, 2$.
For given databases db_1, db_2 on S_1, S_2, selectors sel_1, sel_2 and the corresponding views, two functions f_1, f_2 match if

$f_1(sel(db_1)) \cup_{S_2} db_2 \quad \in SAT(S_2)$ and $f_2(sel(db_2)) \cup_{S_1} db_1 \quad \in SAT(S_1)$.
Two (S_1, S_2, sel_1, sel_2) functions f_1, f_2 are view cooperation functions if the functions match with regard to all $(db_1, db_2) \in (SAT(S_1), SAT(S_2))$.
The problem concerning whether (S_1, S_2, sel_1, sel_2) functions exist is a generalization of the view updateability problem for $S_1 = S_2$ and $sel_1 = sel_2$. In this case, the function f_1 is an embedding function.
The *global view cooperation problem* determines whether view cooperation (S_1, S_2, sel_1, sel_2) functions exist. The *restricted view cooperation problem* determines whether there exist restricted view cooperation (S_1, S_2, sel_1, sel_2)-functions id, id, i.e. for all $(db_1, db_2) \in (SAT(S_1), SAT(S_2))$ with $sel_i(db_i) = (sel_j(db_j))$ and $i, j \in \{1, 2\}, i \neq j$.
Two views defined on S_1, S_2, sel_1, sel_2 are said to be *consistent* if view cooperation functions exist.
View cooperation and integration can be based on the construction of subtype / supertype hierarchies, e.g., for the integration of conceptual graphs. This approach is based on strong semantics for cardinality constraints. The theory of extended entity-relationship models can be used to derive conditions for view cooperations. It is well known [Tha00] that the subtype/supertype hierarchy has to be consistent with the view cooperation schema.

4.1.2 Application of Cooperation to Multi Database Systems

Distributed database systems are based on local database systems and follow a certain integration strategy. Integration is based on total integration of the local conceptual schemata into a global distribution schema.
Open multi-database systems are a variant of distributed systems with a distribution schema that does not integrate the local systems but supports an identification of the database systems and their data. Database system integration has been tackled on the basis of federated database systems. Their architecture is similar to the one in Figure 4.2. The container systems do not contain any additional programs. The global communication and farming system is a simple transfer system in this case. Federated database systems are distributed database systems which use local database systems for support of global applications. Federated database systems have not yet succeeded in practical applications. The main reason is the technical difficulty. Federated systems have to be supported by sophisticated integrity maintenance, powerful communication and transaction protocols and by systems for automatic decomposition and generation of functionality.
Database farms [YTS+99] are generalizing and extending these approaches. Their architecture is displayed in Figure 4.2. Farms are based on the codesign approach [Tha03] and the information unit and container paradigm:

- *Information units* are generalized views. Views are generated on the basis of the database. Units are views extended by functionality necessary for the utilization of view data. We distinguish between *retrieval information units*

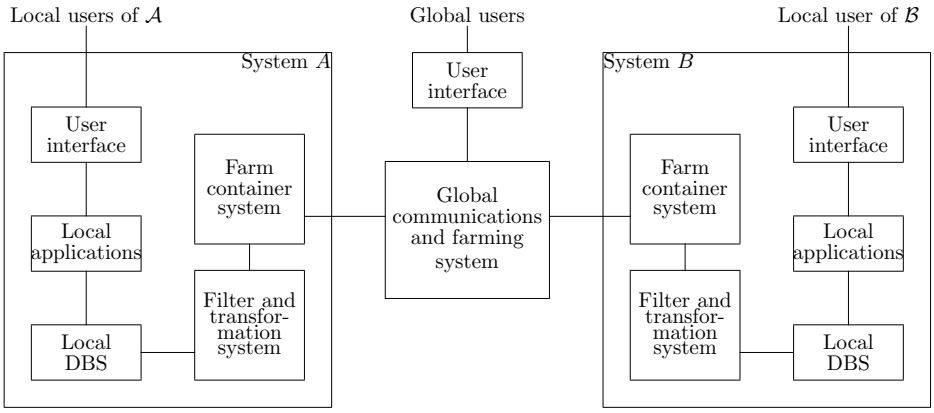


Figure 4.2: Database Systems Farm

and *modification information units*. The first ones are used for data injection. The later ones allow to modify the local database.

- *Containers* support the export and the import of data by bundling information units. Units are composed to containers that can be loaded and unloaded in a specific way.

- The *global communication and farming system* provides the exchange protocols, has facilities for loading and unloading containers and for updates of modification information units.

We do not want to integrate entirely the local databases but provide only *cooperating views*.
Database farms are more complex to design. The computational support is entirely based on classical database technology. Therefore, if we are able to design such integrated system farms the management is feasible.

4.1.3 Application of Cooperation to Incremental Database Systems

Integration of systems can be based on *hub points* at which systems may plug and have the same behavior. Information-lossy integration could be based on abstraction, if the information loss is restricted to those data that is not of interest in the other application or which may be computed by the other application.
The theory of hub types supports *incremental evolution of database systems* [Raa01] which is a specific form of database system evolution. Facility management systems are typical application systems for which incremental evolution could be the ultimate solution. Typical for such applications is the long lifespan of some of the objects. Those objects have a long history of change. We use *auxiliary databases* for support

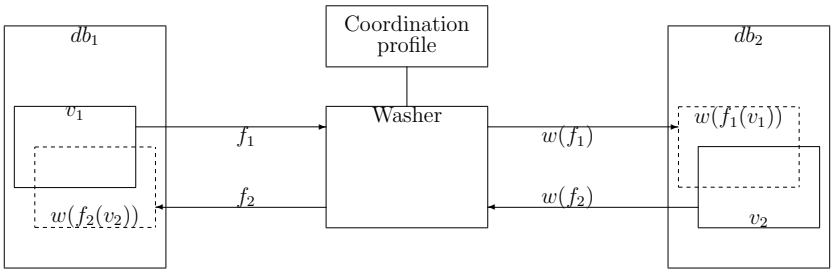


Figure 4.3: The Washer Approach to Collaboration of Databases

of the facility management system. Such data provide help information, information on regulations, information on customers, information on suppliers, etc.
Incremental evolution is thus supported by:

- *Injection forms* enable to inject data into another database. The forms are supported by views and view cooperation approaches. Data injected into another database cannot be changed by the importing database system. The structuring $(\mathcal{S}^{inject}, \Sigma_{\mathcal{S}})$ of the views of the exporting database system is entirely embedded into the structuring $(\mathcal{S}', \Sigma_{\mathcal{S}'})$ of the importing database system. The functionality $(\mathcal{O}^{inject}, \Sigma_{\mathcal{O}})$ of the views of the exporting database system is partially embedded into the functionality $(\mathcal{O}', \Sigma_{\mathcal{O}'})$ of the importing database system by removing all modification operations on the injected data. These data can only be used for retrieval purposes.

- *Insertion forms* enable in insertion data from the exporting database into the importing database. These data can be modified. The structuring $(\mathcal{S}^{insert}, \Sigma_{\mathcal{S}})$ and the functionality $(\mathcal{O}^{insert}, \Sigma_{\mathcal{O}})$ of the views of the exporting database system are entirely embedded into the structuring $(\mathcal{S}', \Sigma_{\mathcal{S}'})$ and the functionality $(\mathcal{O}', \Sigma_{\mathcal{O}'})$ of the importing database system.

4.1.4 Database Collaboration in the Washer Approach

The Cottbus database and information systems research group developed in one of its industry projects [RT99] a specific extension of the view cooperation approach: The *Washer*[1] approach is based on view cooperation and explicit modeling of coordination among several databases. The general architecture is depicted in figure 4.3. The washer is a tool that manages the collaboration based on the the coordination profile. The coordination profile is specified by a *coordination contract*, a *coordination workspace*, *synchronization profile*, *coordination workflow*, and *task distribution*.
Coordination is based on a coordination contract. The *contract* consists of

[1] A washer is a ring of metal between a nut and a bolt, or between two pipes to make a better and tighter joint.

- the coordination party characterization, their roles, rights and relations,

- the organization frames of coordination specifying the time and schema, the synchronization frame, the coordination workflow frame, and the task distribution frame,

- the context of coordination, and

- the quality requirements (ubiquity, security, interpretability, consistency, view consistency, scalability, durability, robustness, performance) for coordination.

We distinguish between the frame for coordination and the actual coordination. Any actual coordination is an instance of the frame. Additionally, it uses an infrastructure. The contract specifies the general properties of coordination. Several variants of coordination may be proposed. The formation of a coordination may be based on a specific infrastructure. For instance, the washer may provide a workspace and additional functionality to the collaborating partners.
Collaboration is based on *communication*, *cooperation*, and *coordination*. Cooperation specification follows a similar approach. It is restricted by the *cooperation contract* that specifies

- the services provided, i.e., informational processes consisting of *views* of the source databases, the *services manager* supporting functionality and quality of services, and the *competence of a service* manifested in the set of tasks that may be performed, and

- requirements for *quality of service*.

Communication contracts specify the collaboration architecture and the style of exchange. Typical collaboration architectures are for example proxy collaboration, broker-customer, or publisher-subscriber collaboration. The exchange frame generalizes protocols and is defined by

- *collaboration style* specifying the supporting programs, the style of collaboration and the collaboration facilities, and

- *collaboration pattern* specifying the roles of the partners, their responsibilities, their rights and the protocols they may rely on.

In a similar fashion we could specify the communication profile and the cooperation profile. The project has led to an integration of SAP R/3, of several Oracle databases and of OLAP functionality of SAS.

4.2 Transformation

The last section described database integration and collaboration strategies in general. Using database collaboration we can define an architecture of converters for collaborating content warehouses that support transformation between similar (but different in detail) schemata.
The *transformation* is based on the extraction of a logic theory supporting reasoning on name spaces between the types of the schemata under consideration. Name spaces of the schemata under consideration may be compared by their "similarity" on the basis of synonym and homonym equalities and inequalities. Equalities and inequalities are enriched by *plausibility coefficients* that measure the confidence of the actual equality or inequality. The confidence measure obeys properties of t-norms used for Fuzzy logics. Additionally we may use *context* for the confidence measure. The logical theory uses a number of comparison predicates:

- *Synonyms* $S \approx T$ specify that two names in schemata under consideration have the same meaning or semantics. Synonyms may also be based on identification expressions. Identification of things could be different in different applications although objects relate to the same thing of the reality, e.g., student number and student identification data.

  Synonym associations can also be developed for query expressions defined on two schemata. Typically, such synonym expressions are semantic conversions for domains, e.g., converting fuel consumption data used in Germany to fuel consumption data used in the US.

  Synonym expressions may be generalized to *data integration mediators*. These can be stored in a database that extends the current database by integrating mediators. We use a similar mechanism on the basis of *extended identification*.

  Synonym associations may be combined with a *preference rule* stating what type name is going to be used if the two synonyms are mapped to one type in the integrated schema.

- *Homonyms* $S \between T$ describe structural equivalence combined at the same time with different meaning and semantics. Homonyms may be simply seen as the negation or inversion of synonymy. Since the confidence level may be different we prefer to use homonyms as well.

- *Hyponyms* and *hypernyms* $S \preccurlyeq T$ hint on subtype associations among types under consideration. The type T can be considered to be a more general type than S and the integration of the two types leads to an explicit subtype association in the integrated schema.

- *Overlappings* and *compatibilites* $S \overset{\exists}{\Cup} T$ describe partial similarities. These similarities can be treated by introduction of generalizing supertypes.

- *Explicit representation conflicts* support the application of conflict resolution strategies such as renaming, homogenizing representations, homogenizing types, application of extended database operations such as extended join and other homogenizing operations. Representation conflicts could lead to integration *obligations* for interactive resolution of those conflicts.

- Abstraction similiarities $\mathfrak{S} \approx \mathfrak{T}]$ support the development of name space transformations. Sub-schemata could be abstracted into abstract types. Abstract types of different schemata could be associated. We may explicitly use this meta-similarity or meta-heterogeneity for association or separation of subschemata.

Additionally, we may require that the equality logic is invariant with the structuring given in the schemata. The metrics can be structurally based. The deduction system in [Tha00] for inclusion and exclusion constraints may easily be generalized to a deduction system for reasoning on synonyms, homonyms, hyponyms and potential supertypes. Since the logical system could be become too granular we may use a threshold logic for more abstract reasoning on potential integrability.
Structural transformation also removes *pragmatic assumptions* made during database development. Typical such assumptions are the *objectification* and the introduction of entity types, relationship types, cluster types, and attribute types. In one application, for instance, it might be useful to use entity types instead of attribute types because of limitations of the platform. If the modeling methodology uses the 'dividing range' then the objectification decision is explicit. Older methodologies do not use this conception and/or use only atomic attributes. In the latter case, objectification has to be based on the elicitation of additional knowledge on the application area.
The structural and semantical information can be used for *preintegration* and *comparison* of schemata. Preintegration is based on a strategy of the order of integration. The best order known so far is the inductive order following the order of structural types, i.e., starting with domains, followed by attribute types, followed by entity types and then finally by relationship types depending on their order. During comparison, naming and structural affinities and conflicts are derived, synonyms are unified to common names, homonyms are separated based on name extensions (e.g., prefixes), hyponyms are used to form hierarchies within the schema, and overlappings are used for developing generalizations.
The result of the step is a *mediating and separating ontology* of types used in the schemata under consideration. This ontology is associated with queries for extraction of the corresponding concepts of the databases. It supports the derivation of mappings for cooperating views.
This mediating ontology can be extended to generation of the global schema $\mathcal{G}$ for the triple $\mathfrak{I} = (\mathcal{G}, \mathfrak{S}, \mathcal{M})$, the collection $\mathfrak{S}$ of local database schemata $\mathcal{S}$ (over a language $\mathcal{A}_{\mathcal{S}}$), and the mapping between $\mathcal{G}$ and $\mathfrak{S}$. Given an equality and inequality theory $\Gamma_{\approx,\emptyset,\preccurlyeq,\overset{\exists}{\cup}}$. We can derive the weakest similarity relation $\backsimeq$ expressing that two

types are definitely equal and the strongest similarity relation $\approxeq$ expressing that two types are potentially equal, i.e., there is no objection against their equality. These two similarity relations may be used for automatic derivation of the

- *weakest global integration schema* $\mathcal{G}_{\preceq} = (\bigcup \mathfrak{S})_{|\preceq}$ and
- *strongest global integration schema* $\mathcal{G}_{\approxeq} = (\bigcup \mathfrak{S})_{|\approxeq}$.

The mappings $\mathcal{M}_{\preceq}$ and $\mathcal{M}_{\approxeq}$ are constructed in a similar way if a preference relation for the choice of the integration type is provided.
The *intermediate global schema* $\mathcal{G}$ is usually a schema that is weaker than the strongest integration schema and stronger than the weakest integration schema. The decisions which strictness for integration is applied will be derived in the following steps.

4.2.1 Development of the Content Warehouse Kernel

The general architecture of the content warehouse uses the separation of source schemata $\mathcal{S}$ into

- sub-schemata $\mathcal{V}(\mathcal{S})$ that must be integrated and sub-schemata that coexist together with other sub-schemata of other applications and
- sub-schemata $\mathcal{S} \setminus \mathcal{V}(\mathcal{S})$ that are not under consideration for integration.

Content warehouses integrating several applications contain all data of the applications. The data is separated into data that belongs to one and only one application and data that belongs to several applications. The relation among the commonly shared data forms the skeleton of the application. This skeleton may be rather complex. We may simplify the skeleton by developing a general integration kernel and by associating the integration kernel with the data that are commonly used. In the simplest case the skeleton forms a star structured content warehouse as pictured above. The kernel coincides with the data structure of the integrated data structure of the content warehouse. This simple structure is often not achievable. We may, however, use a *surrogate kernel* by introducing a number of artificial types that may not have a meaning in the application but nicely support integration and consistent management of data in the content warehouse.
The development of the content warehouse kernel for integrating n tools is based on a number of steps:

- Development of abstractions within the schemata: Since large schemata are hard to understand, we simplify the schema by applying schema abstraction and clustering techniques. Clustering is a recursive procedure that constructs shells of main types. Depending on the adherence, types of a shell may be clustered to one type for external representation and collaboration issues.

- *Development of a hub meta-structure and simplification of the skeleton:* The skeleton of the integrated schema is reconsidered for detection of generalizable structures, for transfer of bilateral associations to trilateral, then of trilateral associations to 4-lateral etc. and finally of (n-1)-lateral associations to n-lateral. If the i-lateral association of source schemata can be easily maintained then we do not transfer them to higher laterality. The transfer to higher associations is based on the introduction of surrogate identifiers that can be mapped to the identification of types. These surrogate identifiers imply two surrogate functional dependencies relating the surrogate with the original identification. The derivation of surrogates is recorded for automatic derivation of integrity constraint management of the surrogate functional dependencies. This transformation of the content warehouse schema forms a hub-like skeleton. This skeleton has the advantage that changes of data by one tool can easily mapped to changes of data for other tools.

- *Development of a surrogate generation facility:* Surrogate identifiers can be simpler handled if the generation mechanism is well-specified. The generation mechanism may be used for derivation of indexes, for derivation of direct search facilities, and for collection of data suites for transfer of data from the warehouse to the tools.

- *Development of version management:* Version management becomes a major obstacle if the local applications are intensively used from time to time. We can integrate the version mechanism into the surrogate value generation if versioning is going to be hierarchical. In this case we use an identifier suffix extension as the version number.

4.2.2 Derivation of Requirements for Wrappers

Finally, requirements for *wrappers* that support integration, import and export of data are developed. These wrappers support the following tasks:

- *Input/output/modification interfaces:* Content in the content warehouse and data used, modified, and generated by local applications must be constantly harmonized. The import/export of data from the warehouse to the tools is based on tool enactment. Whenever an external system requests a data suite necessary the local tool is either transferred from the warehouse or a new data suite for the external system is automatically copied to the warehouse.

- *Consistency control for data suites:* Data suites are kept together for all active external systems. Consistency of kernel data common to all applications is constantly maintained by triggers and monitors observing the state of the database.

- *Consistent playout of data generated by different external systems:* A challenge to the content warehouse that can only be partially resolved is the consistent

playout of data suites by several external systems. To support this challenge, special playout wrappers may be developed that show the results of simulations of several tools. This solution is not a general one but works only under some restrictions applied to the local applications.

The wrappers thus support the hub-based integration within a content warehouse. At the same time wrappers may be used to support consistency management. Consistency management and constraint enforcement are one of the most difficult database design and database development issues. Integrity enforcement is usually supported by

- *decomposition (normalization) of structures* to such structures which allow a simple integrity enforcement (mainly on the basis of keys, referential integrity constraints, and domain constraints), or by
- *extensions of database operations* that maintain consistency of the database while have the same effect as the operation that has been extended, or by
- *special programs for integrity enforcement* such as triggers, assertions and stored procedures, or by
- *transaction management* that rejects all those modifications of the database that leads to inconsistent states, or by
- *application interfaces* keeping consistency of those data suites which may be falsified within a program that uses insert, delete or update operations.

All five approaches have their disadvantages. Decomposition approaches may fail due to the complexity of the integrity constraint set. Extensions of operations cannot be computed in all cases. Special programs supporting integrity management must be combined with control and scheduling facilities in order to avoid avalanches. A solution for the last problem is not known yet. Transaction management might be too restrictive and too pessimistic. The last approach to integrity management is feasible as long as the application is modifying the database only through the application interfaces and the interfaces are well developed.

4.2.3 Collaboration Warehouses

Objects may be developed by each application on their own. Objects belonging together to one version of the development process are called an *object suite*. A *suite* consists of a set of elements, an integration or association schema and obligations requiring maintenance of the association. A suite will be accepted by the collaborating database set

- by transforming the suite into a set of objects within the content warehouse,
- by adding to the identification of objects their object suite identification,

- by extracting the identification tree of the suite identification, and
- by associating the object suite with the application working so far with the suite.

The identification tree of an object suite is called *hub kernel*. Through this hub kernel the object suite can be reestablished.
An external system may either call an existing object suite or insert a new object suite into the content warehouse. If an application calls an object suite and the object suite has been developed by another application then the object suite will be also associated with the new application. Any application may directly change those parts of the object suite which exclusively belong to the structures and attributes of this application. If data in an object suite are changed that belong to several applications then an explicit cooperation function must support the consistency of the parts of the object suite. The hub kernel may not be changed by any application. It is only possible to delete the entire object suite if none of the applications is using one the objects in the suite.
The collaborating database suite is based on an explicit specification of the *cooperation pattern*, of the *coordination pattern* and of the *communication pattern* of object suites. These patterns are composed to a general *collaboration contract*. This contract specifies which application may perform which modification of an object suite under which conditions at which time.

4.3 Presentation as Data Exchange

One special case of data exchange is the export and import of data through user interfaces (UI). Many applications treat user interfaces different from data exchange between applications because data structuring requirements are different. This leads to inconsistencies while comparing UI behavior and application programming interfaces and makes UI interaction hard to automate. The reason for this behavior is a poor separation between application logic (workflows), communication logic (interaction stories), and presentation rendering. A general goal of developing information systems in general and content warehouses in particular is violated: content should be available independently from the underlying infrastructure that is used for communication.
From the point of view of the content warehouse both data exchange and user interaction share a common ground: both processes are dialogs in a linguistic sense. The partners change their roles between "speaking" and "listening" — in terms of applications between "sending content" and "receiving content". So there is no need for a content warehouse to distinguish between different communication partners on this conceptual level. External applications as well as the presentation machine for rendering the user output share a common interface: data suites based on the theory of media objects (see chapter 5 or [ST01]).

5 Interaction

Interaction describes mutual impact between two (or more) partners. In terms of information systems interaction modeling expresses the way of data transfers between systems, applications, or human beings.
Application programming interfaces (APIs) of information systems are often organized in a request-response manner: an external application sends a request to the information system, the request is processed by the information system and the result of this processing is transfered to the application.
When modeling user interfaces (UI) the wizard approach is often favored: due to the limited receptivity of human beings the request is chopped into pieces and communication takes place as a dialog sequence. The "wizard" guides the user through the dialogs until the request is finished.
There is actually no need to separate between APIs and UIs on a functional level in modeling interaction. Data transfer between applications can be as dialog sequences, while offering data to a user is in fact exchange of data suites with a certain structure.
This chapter describes SiteLang ([Tha03]), a modeling language for expressing interaction based on storyboards. It is demonstrated that Sitelang can be used on different abstraction layers during system development. Additionally, this chapter introduces a rapid prototyping environment that generates executable code from SiteLang specifications.

5.1 Storyboarding: Modeling Interaction

According to [exp06] storyboarding is the process of producing sketches of the shots of a script. It helps to think about how a film is going to look. Storyboarding is especially useful for complex visual sequences e.g. elaborate shots or special effects sequences. Sometimes a film only uses storyboards for difficult sequences; other times the entire film is storyboarded. Pictures within a storyboard contain markers (such as arrows or frames) to denote movements or camera pannings. Pictures are connected with transitions. There is no fixed rule what a certain picture should contain, a scene can be drawn within one picture or within a sequence of pictures.
The methodology of storyboarding can be adapted to interaction modeling. A storyboard is characterized by:

- A set of *actors*: actors describe groups of similar users and can be represented as concepts $\mathfrak{A}$ as described in chapter 3. For each actor exist
  - a profile of properties,

  - a portfolio describing tasks that have to be fulfilled by the actor in certain situations together with operational parameters (priority, duration of activities, etc.),
  - security profile describing the access rights of the actor, and
  - goals to be fulfilled.

- *Content objects* (also called media objects) connect the interaction to the data within the information system.

- A *story* is a plot of a narrative work and consists of several scenes and transitions between scenes. The storyspace is composed of all stories.

- A scene is a part in a story characterized as a sequences of uninterrupted action. Scenes are the basic composition units of interaction. Each scene is associated with a set of actors and a set of content objects.

- A set of *scenarios*: stories can be played in different scenarios. A scenario is a walk through the story. Scenarios are composed by scenes.

- The basic unit of interaction are dialogs. Each scene consists of several dialogs. Transitions between dialogs are made by dialog steps.

- A dialog step describes the transition from one dialog to another. Dialog steps are triggered on certain events. They are guarded by preconditions that must hold before the dialog step is executed. Postconditions ("accept on") can be defined, too. If a postcondition of a dialog step is not fulfilled the actions that took place during execution of the dialog step are rolled back. During the execution of a dialog step actions on the content objects depending on transient data from the dialogs are done.

- A walk through a scenario by a set of collaborating agents is called a scenario instance. The sequence of scenes visited by agents at certains point in time is called a run of the scenario.

Figure 5.1 shows the graphical representation of a story specification for the process of applying a business trip. Scene "Application" is refined in figure 5.2 by dialog steps.

5.2 Interaction Prototyping

Developing interaction components, especially in the case of user interaction is a challenging task because requirements are often "soft" and cannot be formalized. Design and implementation of graphical interfaces is very cost intensive and time consuming so there is a need for early validation of interaction specifications to reduce the impact of errors and misunderstandings. Because central requirements

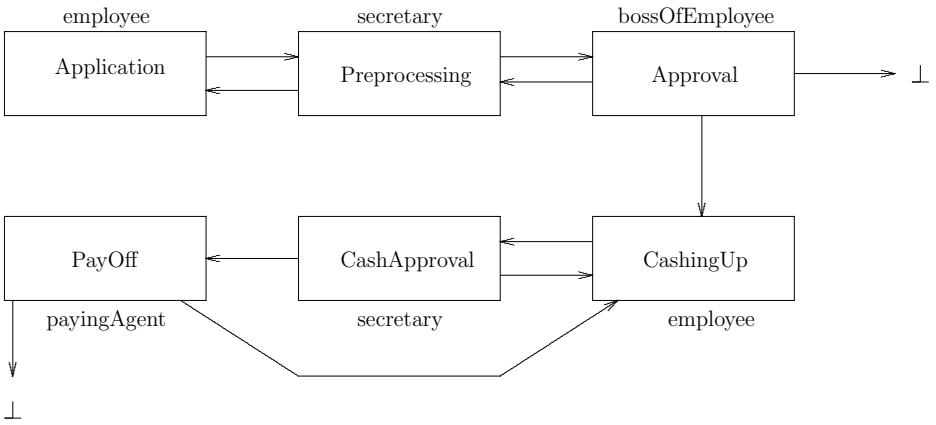


Figure 5.1: Business Trip Application

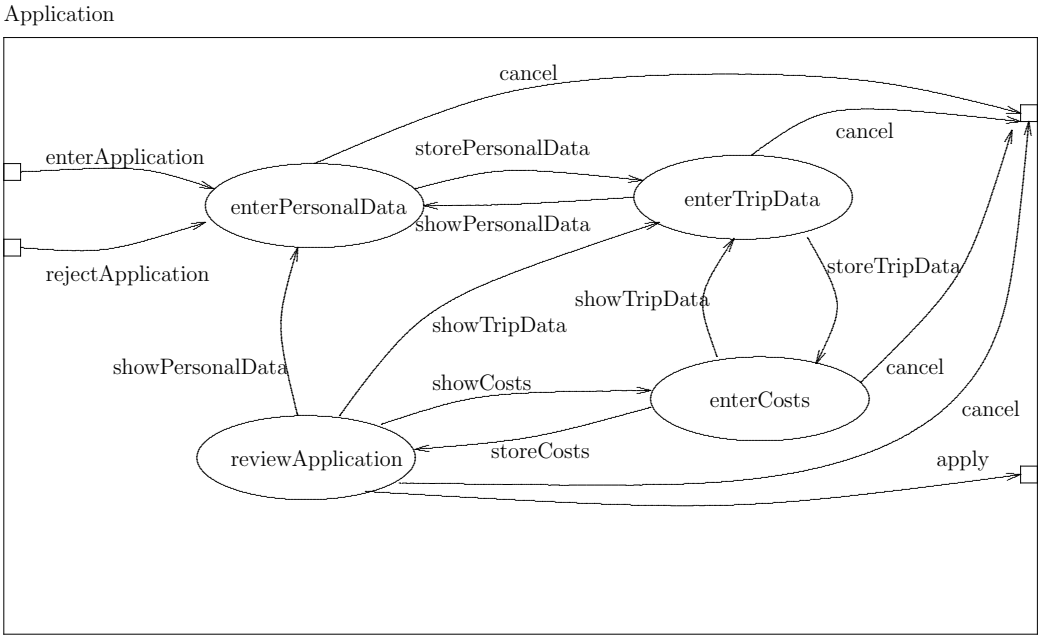


Figure 5.2: Refinement of Scene “Application” with Dialog Steps

are only informal there is no possibility to do this by a formal proof. The interface has to be simulated and presented to test users. Their comments are taken into account for improving the interaction model.

5.2.1 Executable Specifications and Rapid Prototyping

Information systems are usually developed using an abstraction layer model. Simulations have to be made as early as possible. Because the usage context of the interface directly influences the user's perception, the simulation has to take place on the target platform that is used for production. Thus, the specification has to be made executable. Executable specifications need a formal translation between the languages on different abstraction levels. Additionally, every translation between two abstraction layers that is made by hand is lost for documentation purposes and leads to the construction of legacy systems, because following developers are usually not able to understand such breaks. A golden rule of system development based on abstraction layers is: a fact that was modeled in a step i during the development process must be visible in steps $j > i$.
Unfortunately, on development process layers higher than the implementation layer, many specification details necessary for execution are still not present. So, another facility is needed: a framework for rapid prototyping. During rapid prototyping missing parts of the executable specification are automatically replaced by defaults to render a complete interface specification on the implementation level which can be transfered to the target system.
The resulting development cycle was sketched in [FSBTS04]:

- The development begins with the specification of the use cases that describe the general requirements for the interface.
- Use cases are replaced by a coarse storyboard describing the general interface flow.
- In an iterative process the interface is refined until all dialogs are defined. At each step formal defined properties are validated and simulations are made.

5.2.2 Using SiteLang for Code Generation

SiteLang suits very well for this style of interface specification because it's general syntax allows specifications of different granularity. Based on the fixed semantics of SiteLang transformation for different target platforms can be derived. As an example, figure 5.3 shows the tool support for generating Java Servlet based Web applications compiled from SiteLang specifications.
A graphical editor creates SiteLang specifications stored as in a XML file. A complete SiteLang specification consists of

- scenes with dialog steps and actor definitions

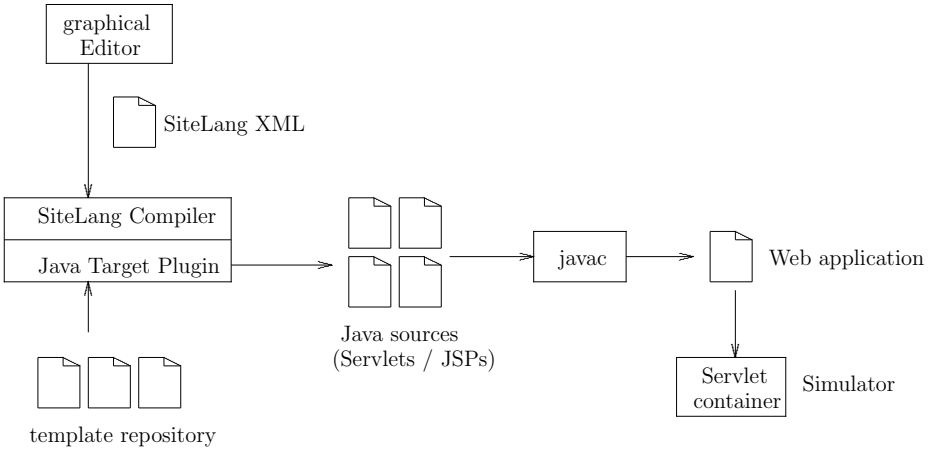


Figure 5.3: Tool Support for Interface Development Based on SiteLang

- dialogs
- scenarios with scene instantiations (A scene instantiation is a renamed scene specification. Thus, a predefined scene can be used twice in the same scenario.)
- events (An event is a data structure that is passed between the interacting partners.)
- type definitions for content objects
- Resource definitions that map content objects, actors, dialogs and other facets to real resources (databases, screen masks, etc.) in the resulting application.

A minimal SiteLang specification the compiler can successfully transfer to a executable specification is a set of scenes with simple dialog steps and a scenario specification that mounts scene specifications together. All other facets can be replaced by defaults and refined later. For example, the rapid prototyping component creates dialog masks automatically based on available templates ("style guides"). Input masks for data structures (events) are generated automatically. This makes dialog masks "look more realistic" during testing even on early stages of interface development and enhances the acceptance of the interface by the testing users.
During compilation the SiteLang compiler for the Java Servlet application target platform generates classes for

- the *management of stories*: the story class manages meta information for a specified SiteLang story (which scenes, which possible actors, etc). The extension of the class contains all active story instances. A story instance contains information which scene instance is currently active, which actor is currently associated with which user. Additionally, a scene instance contains local variables as part of the system state.

- the *management of scenes*: for each scene a separate class derived from a common super class is created. These classes provide meta information for the scene (which dialog steps, which actors, access to the content object class, links to dialog definitions, ...). The extension of the class contains all active scene instances of this class. These instances provide the state of the scene, current actor-to-user mappings, and the content object instance.

- the *management of dialog steps:* dialog steps are represented as members of scene instances. Events are propagated to the corresponding scene instance. For this instance the trigger conditions for each dialog step are evaluated and the dialog step is activated if necessary.

- the *management of content objects:* each content object is translated to a specific class derived from a common super class. The content object classes provide meta information about the structure, the functionality, the micro behavior, and the associations of the corresponding content object. Additionally, the generic database load / store mechanisms are provided. Each class instance contains the values for a concrete content object instance usable in dialog step actions.

- the *management of actors and users:* for each defined actor a class derived from a common super class is defined which implements this actor model. This class can be used as an oracle that can be asked for certain properties of the actor. Additionally, this class manages mappings between actors and users.

Bibliography

[ACM97] Serge Abiteboul, Sophie Cluet, and Tova Milo. Correspondence and Translation for Heterogeneous Data. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 1997.

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BHvdT05] Guido Boella, Joris Hulstijn, and Leendert W. N. van der Torre. Argumentation for Access Control. In *AI*IA*, pages 86–97, 2005.

[BM99] Catriel Beeri and Tova Milo. Schemas for Integration and Translation of Structured and Semi-structured Data. In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 1999.

[BR01] Phil A. Bernstein and Erhard Rahm. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10:334–350, 2001.

[BvdT05] Guido Boella and Leendert W. N. van der Torre. Permission and Authorization in Normative Multiagent Systems. In *ICAIL*, pages 236–237. ACM, 2005.

[CDSS98] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 177–188. ACM Press, 1998.

[CGL+98] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, and Riccardo Rosati. Information Integration: Conceptual Modeling and Reasoning Support. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City,*

New York, USA, August 20-22, 1998, Sponsored by IFCIS, The Intn'l Foundation on Cooperative Information Systems, pages 280–291. IEEE-CS Press, 1998.

[Con86] B. Convent. Unsolvable problems related to the view integration approach. In *ICDT'86*, volume 243 of *Lecture Notes in Computer Science*, pages 141–156. Springer, 1986.

[CP98] Stefano Spaccapietra Christine Parent. Issues and Approaches of Database Integration. *CACM*, 41(5):166–178, 1998.

[CSS99] Stefan Conrad, Gunter Saake, and Kai-Uwe Sattler. Informationfusion - Herausforderung an die Datenbanktechnologie. In *Proc. BTW*, pages 307–316. Springer, 1999.

[exp06] exposure.co.uk. Guide to Film-Making Website. http://www.exposure.co.uk/eejit/storybd/, March 2006.

[Fit91] Melvin Fitting. Many-valued modal logics. *Fundam. Inform.*, 15(3-4):235–254, 1991.

[Fit92] Melvin Fitting. Many-Valued Model Logics II. *Fundam. Inform.*, 17(1-2):55–73, 1992.

[Fit00a] Melvin Fitting. Databases and Higher Types. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 2000.

[Fit00b] Melvin Fitting. Modality and Databases. In Roy Dyckhoff, editor, *TABLEAUX*, volume 1847 of *Lecture Notes in Computer Science*, pages 19–39. Springer, 2000.

[FKST00] T. Feyer, O. Kao, K.-D. Schewe, and B. Thalheim. Design of data-intensive web-based information services. In Q. Li, Z. M. Özsoyoglu, R. Wagner, Y. Kambayashi, and Y. Zhang, editors, *WISE 2000, Proceedings of the First International Conference on Web Information Systems Engineering, Volume I (Main Program), Hong Kong, China, Jun 19-21, 2000*, pages 462–467. IEEE Computer Society, 2000.

[FSBTS04] Gunar Fiedler, Thomas Schwanzara-Bennoit, Bernhard Thalheim, and Peggy Schmidt. State-, HTML-, and Object-Based Dialog Design for Voice Web Applications. In Maristella Matera and Sara Comai, editors, *Engineering Advanced Web Applications*. Rinton Press, 2004.

[GBD04] Georg Gottlob, András A. Benczúr, and János Demetrovics, editors. *Advances in Databases and Information Systems, 8th East European Conference, ADBIS 2004, Budapest, Hungary, September 22-25, 2004, Proceesing*, volume 3255 of *Lecture Notes in Computer Science*. Springer, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GW98] B. Ganter and R. Wille. *Formal concept analysis - Mathematical foundations*. Springer, Berlin, 1998.

[HC01] George T. Heinemann and William T. Councill. *Component-Based Software Engineering*. Addison-Wesley, 2001.

[HMW05] Volker Haarslev, Ralf Möller, and Michael Wessel. Description Logic Inference Technology: Lessions Learned in the Trenches. In Ian Horrocks, Ulrike Sattler, and Frank Wolter, editors, *Description Logics*, volume 147 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[Hor98] Ian Horrocks. The FaCT System. In Harrie C. M. de Swart, editor, *TABLEAUX*, volume 1397 of *Lecture Notes in Computer Science*, pages 307–312. Springer, 1998.

[HR01] Theo Härder and Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Springer Verlag, second edition, 2001.

[HRU96] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD '96*, pages 205–216, Montreal, June 1996.

[KBF+05] Tatjana Kruscha, Björn Briel, Gunar Fiedler, Kai Jannaschk, Thomas Raak, and Bernhard Thalheim. Integratives HMI-Warehouse für einen durchgängigen HMI-Entwicklungsprozess. In VDI, editor, *Elektronik im Kraftfahrzeug 2005. 12. Internationaler Kongress Electronic Systems for Vehicles*, number 1907 in VDI-Berichte. VDI, VDI-Verlag, 2005.

[Koz00] Dexter C. Kozen. *Automata and Computability*. Springer, 2000.

[Len02] Maurizio Lenzerini. Data integration: a theoretical perspective. In ACM, editor, *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2002: Madison, Wisconsin, June 3–5, 2002*, pages 233–246, New York, NY 10036, USA, 2002. ACM Press.

[LG99] Bertram Ludäscher and Amarnath Gupta. Modeling Interactive Web Sources for Information Mediation. In Peter P. Chen, David W. Embley, Jacques Kouloumdjian, Stephen W. Liddle, and John F. Roddick, editors, *Advances in Conceptual Modeling: ER '99 Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling, Paris, France, November 15-18, 1999, Proceedings*, volume 1727 of *Lecture Notes in Computer Science*, pages 225–238. Springer, 1999.

[LRL98] T. W. Ling, S. Ram, and M.-L. Lee, editors. *Proc. 17th Int. ER Conf., Conceptual Modeling - ER'98*, LNCS 1507, Singapore, Nov. 16 - 19, 1998, 1998. Springer, Berlin.

[McA00] David A. McAllester, editor. *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*. Springer, 2000.

[MCGF99] M. Catherine McCabe, Abdur Chowdhury, David A. Grossman, and Ophir Frieder. A Unified Environment for Fusion of Information Retrieval Approaches. In *Proceedings of the 1999 ACM CIKM International Conference on Information and Knowledge Management, Kansas City, Missouri, USA, November 2-6, 1999*, pages 330–334. ACM, 1999.

[Min88] J. Minker, editor. *Foundations of deductive databases and logic programming*. Morgan Kaufmann, San Mateo, 1988.

[Moo01] D. L. Moody. *Dealing with complexity: A practical method for representing large entity-relationship models*. PhD thesis, Dept. of Information Systems, University of Melbourne, 2001.

[Ont] http://www.ontopia.net/topicmaps/.

[PAN06] PANGAEA. Publishing Network for Geoscientific & Environmental Data. http://www.pangaea.de, March 2006.

[Paw73] Z. Pawlak. Mathematical foundations of information retrieval. Technical Report CC PAS Reports 101, Warszawa, 1973.

[Pe01] S. Pepper and G. Moore (eds.). XML topic maps (XTM) 1.0. http://www.topicmaps.org/xtm/1.0/, 2001. TopicMaps.Org.

[PV99] Yannis Papakonstantinou and Pavel Velikhov. Enhancing Semistructured Data Mediators with Document Type Definitions. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Austrialia*, pages 136–145. IEEE Computer Society, 1999.

[Raa01] T. Raak. Database systems architecture for facility management systems. Master's thesis, FHL, Civil Engineering Dept., Cottbus, 2001.

[RT99] S. Radochla and B. Thalheim. Umstrukturierung eines Data-Warehouse in ein effizientes Decision Support System. In F. Mäurer F. Hüsemann, K. Küspert, editor, *Jenauer Schriften zur Mathematik und Informatik*, volume Math/Inf/99/16, pages 92 – 96, Jena, 1999.

[Seh03] H.W. Sehring. *Konzeptorientiertes Content Management: Modell, Systemarchitektur und Prototypen*. PhD thesis, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, 2003.

[Sow00] John F. Sowa. *Knowledge Representation, Logical, Philosophical, and Computational Foundations*. Brooks/Cole, a division of Thomson Learning, Pacific Grove, California, 2000.

[SS03] J.W. Schmidt and H.-W. Sehring. Conceptual content modeling and management - The rationale of an asset language. In *Proc. PSI'03, LNCS , Springer, 2003*, 2003. Perspectives of System Informatics.

[SS04] Hans-Werner Sehring and Joachim W. Schmidt. Beyond Databases: An Asset Language for Conceptual Content Management. In Gottlob et al. [GBD04], pages 99–112.

[ST01] K.-D. Schewe and B. Thalheim. Modeling Interaction and Media Objects. In M. Bouzeghoub, Z. Kedad, and E. Métais, editors, *NLDB. Natural Language Processing and Information Systems, 5th Int. Conf. on Applications of Natural Language to Information Systems, NLDB 2000, Versailles, France, Jun 28-30, 2000, Revised Papers*, volume 1959 of *LNCS*, pages 313–324. Springer, 2001.

[ST04] Peggy Schmidt and Bernhard Thalheim. Component-Based Modeling of Huge Databases. In Gottlob et al. [GBD04], pages 113–128.

[Sys] Racer Systems. Racer Pro. http://www.racer-systems.com.

[TH02] Klaudia Hergula Theo Härder. Ankopplung heterogener Anwendungssysteme an Föderierte Datenbanksysteme durch Funktionsintegration. *Informatik - Forschung und Entwicklung*, 17:135–148, 2002.

[Tha00] B. Thalheim. *Entity-relationship modeling – Foundations of database technology*. Springer, Berlin, 2000. See also http://www.informatik.tu-cottbus.de/∼thalheim/HERM.htm.

[Tha03] B. Thalheim. Informationssystem-Entwicklung - Die integrierte Entwicklung der Strukturierung, Funktionalität, Verteilung und Interaktivität von großen Informationssystemen. Preprint I-2003-15, Cottbus Tech, Computer Science Institut, BTU Cottbus, 21. 9. 2003 2003.

[Tha05] Bernhard Thalheim. Component development and construction for database design. *Data Knowl. Eng.*, 54(1):77–95, 2005.

[TSK06] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Pearson Education, Addison-Wesley, 2006.

[Voo89] Frans Voorbraak. The logic of actual obligation. An alternative approach to deontic logic. *Philosophical Studies*, 55, Issue 2:173–194, 1989.

[Wie95] Gio Wiederhold. Modelling and System Maintanance. In Mike P. Papazoglou, editor, *OOER'95: Object-Oriented and Entity-Relationship Modelling, 14th International Conference, Gold Coast, Australia, December 12-15, 1995, Proceedings*, volume 1021 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1995.

[YTS+99] S. Yigitbasi, B. Thalheim, K. Seelig, S. Radochla, and R. Jurk. Entwicklung und Bereitstellung einer Forschungs- und Umweltdatenbank für das BTUC Innovationskolleg. In F. Hüttl, D. Klem, and E. Weber, editors, *Rekultivierung von Bergbaufolgelandschaften*, pages 269–282. Walter de Gruyter, Berlin, 1999.