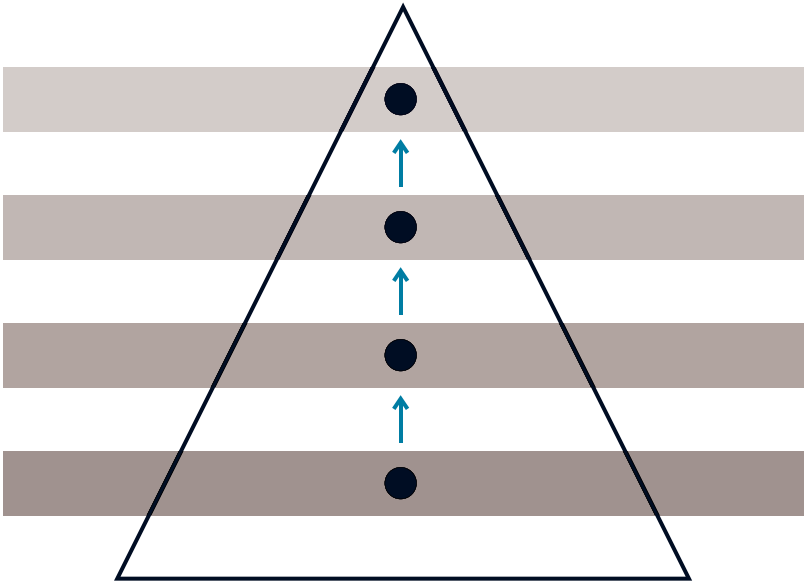
Metamodeling for Method Engineering

2021 Open-access Edition

edited by
Manfred A. Jeusfeld,
Matthias Jarke, and
John Mylopoulos



Metamodeling for Method Engineering

2021 Open-Access Edition

Cooperative Information Systems
Michael P. Papazoglou, Joachim W. Schmidt, and John Mylopoulos, editors

Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering, Nikota K. Kasabov

Advances in Object-Oriented Data Modeling, Michael P. Papazoglou, Stefano Spaccapietra, and Zahir Tari, editors

Workflow Management: Models, Methods, and Systems, Wil van der Aalst and Kees Max van Hee

A Semantic Web Primer, Grigoris Antoniou and Frank van Harmelen

Aligning Modern Business Processes and Legacy Systems: A Component-Based Perspective, Willem-Jan van den Heuvel

A Semantic Web Primer, second edition, Grigoris Antoniou and Frank van Harmelen

Service-Oriented Computing, Dimitrios Georgakopoulos and Michael P. Papazoglou, editors

At Your Service, Elisabetta di Nitto, Anne-Marie Sassen, Paolo Traverso, and Arian Zwegers, editors

Metamodeling for Method Engineering, Manfred A. Jeusfeld, Matthias Jarke, and John Mylopoulos, editors

Metamodeling for Method Engineering

Edited by Manfred A. Jeusfeld, Matthias Jarke, John Mylopoulos

This online edition is published by its editors, (c) 2021

This is the 2021 Open-Access Edition of the book "Metamodeling for Method Engineering" Re-published by its editors under the Creative Commons license CC BY-NC-ND 4.0,see

https://creativecommons.org/licenses/by-nc-nd/4.0/

All rights to this edition held by the editors.
Contact: Manfred A. Jeusfeld, University of Skövde, Box 408, 54128 Skövde, Sweden
email: manfred.jeusfeld@acm.org

last update: 2026-05-28

Copyright notice of the original 2009 book publication:

© 2009 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

MIT Press books may be purchased at special quantity discounts for business or sales promotional use. For information, please email ⟨special_sales@mitpress.mit.edu⟩ or write to Special Sales Department, The MIT Press, 5 Cambridge Center, Cambridge, MA 02142.

This book was set in Times New Roman and Syntax on 3B2 by Asco Typesetters, Hong Kong.
Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Metamodeling for method engineering / edited by Manfred A. Jeusfeld, Matthias Jarke, John Mylopoulos.
p. cm. — (Cooperative information systems)
Includes bibliographical references and index.
ISBN 978-0-262-10108-0 (hc : alk. paper)
1. Programming (Mathematics) 2. Engineering models. I. Jeusfeld, Manfred. II. Jarke, Matthias. III. Mylopoulos, John. IV. Series.

T57.7.M48 2009
003′.3—dc22 2008047203

10 9 8 7 6 5 4 3 2 1

Contents

Online resources for the 2021 Open-Access Edition

The ConceptBase system used as a tool for exercises in this book can be downloaded from

http://conceptbase.cc

under a BSD-style open-source license.

Sources for examples for some of the chapters and other resources such as slides for lectures on metamodeling can be obtained from

http://conceptbase.cc/CB-Resources.html

This replaces the content of the CD-ROM that was accompanying the original 2009 book edition of the textbook. The open-access edition of the textbook features links to the CD-ROM contents on the left margin of the pages. There are three types of links:

source/sources: point to the textual Telos source codes (file extension sml); view those files with a text editor and/or load them to ConceptBase via the tool CBIva

slides: point to slides discussing textbook sections

gel: ConceptBase graph files ready to be loaded by the tool CBGraph; by clicking on them you can download them via your web browser and start them by the ConceptBase tool CBGraph

The sources pointed to by links and the graph files may have slightly updated definitions compared to the source code samples in the textbook since they are being actively maintained.

Further example models for ConceptBase are available from the ConceptBase Forum at

http://conceptbase.sourceforge.net/CB-Forum.html

Since 2026, the CB-Forum is only available to registered users. Registration instructions are at the above link. As registered user, you have also capabilities, e.g. add your own examples.

Series Foreword

The traditional view of information systems as tailor-made, cost-intensive database applications is changing rapidly. The change is fueled partly by a maturing software industry, which is making greater use of off-the-shelf generic components and standard software solutions, and partly by the onslaught of the information revolution. In turn, this change has resulted in a new set of demands for information services that are homogeneous in their presentation and interaction patterns, open in their software architecture, and global in their scope. The demands have come mostly from application domains such as e-commerce and banking, manufacturing (including the software industry itself), training, education, and environmental management, to mention just a few.

Future information systems will have to support smooth interaction with a large variety of independent multivendor data sources and legacy applications, running on heterogeneous platforms and distributed information networks. Metadata will play a crucial role in describing the contents of such data sources and in facilitating their integration.

As well, a greater variety of community-oriented interaction patterns will have to be supported by next-generation information systems. Such interactions may involve navigation, querying, and retrieval, and will have to be combined with personalized notification, annotation, and profiling mechanisms. Such interactions will also have to be intelligently interfaced with application software, and will need to be dynamically integrated into customized and highly connected cooperative environments. Morever, the massive investments in information resources, by governments and businesses alike, call for specific measures that ensure security, privacy, and accuracy of their contents.

All these are challenges for the next generation of information systems. We call such systems *cooperative information systems*, and they are the focus of this series.

In layman's terms, cooperative information systems are servicing a diverse mix of demands characterized by *content—community—commerce*. These demands are

originating in current trends for off-the-shelf software solutions, such as enterprise resource planning and e-commerce systems.

A major challenge in building cooperative information systems is to develop technologies that permit continuous enhancement and evolution of current massive investments in information resources and systems. Such technologies must offer an appropriate infrastructure that supports not only development, but also evolution of software.

Early research results on cooperative information systems are becoming the core technology for community-oriented information portals or gateways. An information gateway provides a "one-stop-shopping" place for a wide range of information resources and services, thereby creating a loyal user community.

The research advances that will lead to cooperative information systems will not come from any single research area within the field of information technology. Database and knowledge-based systems, distributed systems, groupware, and graphical user interfaces have all matured as technologies. While further enhancements for individual technologies are desirable, the greatest leverage for technological advancement is expected to come from their evolution into a seamless technology for building and managing cooperative information systems.

The MIT Press *Cooperative Information Systems* series will cover this area through textbooks and research editions intended for the researcher and the professional who wishes to remain up-to-date on current developments and future trends. The series will publish three types of books:

- textbooks or resource books intended for upper-level undergraduate or graduate-level courses
- research monographs, which collect and summarize research results and development experiences over a number of years
- edited volumes, including collections of papers on a particular topic

Authors are invited to submit to the series editors book proposals that include a table of contents and sample book chapters. All submissions will be reviewed formally and authors will receive feedback on their proposal.

John Mylopoulos
jm@cs.toronto.edu
Dept. of Computer Science
University of Toronto
Toronto, Ontario
Canada

Michael Papazoglou
M.P.Papazoglou@kub.nl
INFOLAB
P.O. Box 90153
LE Tilburg
The Netherlands

Joachim W. Schmidt
j.w.schmidt@tu-harburg.de
Software Systems Institute
Technische Universität
TUHH
Hamburg, Germany

Introduction

This book addresses researchers in the field of conceptual modeling as well as practitioners with advanced modeling skills. It provides the foundations for *method engineering*, that is, the definition of techniques to design software systems, and applies these foundations in five case studies ranging from designing chemical processes to the development of new modeling-language constructs.

The Problem

A computerized software system constitutes a complex reflection of the realities that it models, is part of, and interacts with. Specialized methods have been developed to support the whole life cycle of such a system, from early requirements to installation and maintenance. The methods introduced over the last thirty years include programming paradigms, design techniques, modeling languages, and project management, to name a few.

As software systems become more complex and diversified, so do the development methods we use to build them. Standard methods, such as ones based on the Unified Modeling Language (UML), are not always appropriate for any one development project, since they represent a specific view on the system and reality, which may be in conflict with the requirements of the project. Moreover, new technologies, such as Web services and information quality, are hardly addressed by standard methods.

This book presents a practical approach to *engineering new methods* for building software, founded on construction of *metamodels*. A *method* is regarded here as a set of rules and instructions that guide users of the method in creating models about an information system. The concept of a *model* subsumes any representation of statements about some artifact. For example, the source code of a program is a model, as is a list of requirements. As suggested by the title, defining a method by means of a metamodel is the goal and subject of this book.

Structure of the Book

The book is separated into two main parts plus an accompanying CD-ROM. The first part, consisting of chapters 1 through 3, presents the theory and state of the art of metamodeling for method engineering. Chapter 1 introduces information modeling, which underlies not just the development of a software system, but other activities such as the specification of requirements. Information models capture knowledge about some universe of discourse at a level that is closer to human conceptualizations of the domain than to a computer implementation. The knowledge in an information model can be categorized as concerning static, dynamic, intentional, and social aspects of the system. In a software system, static aspects are usually stored as data and dynamic aspects help the design of transactions, whereas the social and intentional aspects are essential in choosing between design alternatives. Chapter 1 surveys techniques developed for expressing each different category of knowledge, including informal ones, which are accessible to prospective users, and formal ones, which are amenable to machine manipulation and reasoning.

Chapter 2 investigates the potential of metamodeling for software system development. Essentially, a metamodel is a model about a collection of models. Metamodeling has been used to define and extend the capabilities of modeling frameworks. Two main approaches to metamodeling are presented. The first approach, the Information Resources Dictionary System (IRDS) Standard, defines four levels of abstraction: the data level, containing actual data of an application; the model level, containing models of the information system (including source code); the metamodel level, containing definitions of modeling notations and languages; and the metametamodel level, containing the facilities to define modeling languages. Metamodeling is thus the art of designing the contents of the metamodel level, that is, of designing modeling notations and their interrelationships. The second approach is an investigation of key perspectives on information systems development: the goal aspect, the development process aspect, the ontological aspect, and the notational aspect. Typical metamodeling tools, of which chapter 2 describes a few well-known representatives, emphasize a subset of these aspects, depending on the main applications to which they are targeted.

Chapter 3 presents the features of the ConceptBase system as an environment for method engineering. ConceptBase is based on the metamodeling constructs of Telos, a knowledge representation language. The basic concept is a statement (or *proposition*). Concepts like inheritance, class membership, and attribution are all defined on top of statements through logical axioms. Since statements can be formed at any of the four IRDS levels, Telos is an ideal framework for metamodeling. ConceptBase is basically a model management system. Within ConceptBase, metamodeling and method engineering become construction and analysis activities for metamodels.

The chapter applies the functionality of ConceptBase to an instructional example, the Yourdan method for software system development. Well-known modeling languages like entity-relationship diagrams and data flow diagrams can be defined with ConceptBase. The Yourdan method itself is represented by a software process model. This model is part of the metamodel level, along with modeling languages. Particular attention is paid to the construction of internotational constraints, which define the integrity of a given set of models about a software system. Metalevel rules can be defined to capture the semantics of link types in conceptual modeling languages, for example, the transitivity of subclass relations. Finally, some examples on measuring models and modeling processes are given in the chapter in the form of ConceptBase queries.

The second, and larger, part of the book reports on applications of the metamodeling approach to method engineering. All chapters in this part are stand-alone in the sense that they describe a solution to a problem from a specific domain. Chapter 4 reports on a method engineering case study in telecommunications service design. The domain of telecommunications service is characterized by specialized design techniques, but there is a lack of suitable tools to guide the requirements engineering process. The solution proposed in chapter 4 combines three aspects in service design. Intelligence models guide the developer to solutions to telecommunications design problems. Development models contain the standard procedures for service design, including milestone definitions and iterations. Finally, so-called negotiation models interface from domain models to development models. The collection of models is interrelated by rules and constraints. Queries are used to check whether all requirements of a particular service design have been addressed.

Chapter 5 investigates construction aids for hyperbooks. A hyperbook is a complex graph whose navigation structure is crucial for its readability. To allow interoperability, different chunks of a hyperbook distributed over the World Wide Web are described by Resource Description Framework (RDF) statements. It turns out that the logical foundation of Telos makes it an ideal candidate for storing RDF statements and for querying the metadata of a distributed hyperbook. The Telos-based solution is incorporated into a hyperbook system that is used in academic courses.

The interaction of requirements monitoring with a goal model is the subject of chapter 6. The DealScribe system incorporates metamodels of requirements analysis and goal representations. This allows the specification of goals such as "each stakeholder will contribute equally to the requirements dialog during information system development." The assessment of goal fulfillment is used to guide the dialog of developers. During different phases, different goals are monitored. Glimpses into the future are made possible by so-called hypothetical updates to the model repository. Hence, before actually performing an action, a developer can check its consequences

for goal fulfillment. The system also includes a metamodel of linking requirements to express whether one requirement is subsumed by another. Moreover, conflicts among requirements can be expressed and dealt with by voting on alternative resolutions. DealScribe is a working tool that addresses requirements analysis and goal monitoring in a comprehensive way. It uses dedicated modeling languages for expressing requirements and goals and implements these languages using Telos metamodels.

Chapter 7 returns to a more theoretical metamodeling topic: the definition of semantic abstraction principles, such as specialization, attribution, class membership, and part-of relations. Interestingly, there is an abstraction principle that can be positioned between class membership (an object is an instance of a class) and class specialization (one class is more special than another class). This abstraction principle is called materialization. It allows expression of the idea that some attributes of a class are specialized (or inherited), whereas others are instantiated. A class model defines attributes that are shared by all instances of a particular class (e.g., the gas mileage of a certain car model). A class materialization defines properties that are specific to the instances of a particular class (e.g. the serial number of a car). The new abstraction principle ''materializes'' is defined by metalevel formulas. Active rules are used to propagate certain class model properties to the class instances (e.g., the gas mileage is propagated to the car instances that conform to a certain model). The materialization relationship itself is subject to querying, for example, to find out all car classes that materialize a certain car model. The chapter shows that software developers need not be restricted by predefined semantic abstraction principles as provided by a standard modeling language, such as UML. Instead, they can engineer their own abstraction principles on demand. The examples in the chapter are included on the accompanying CD-ROM.[1]

Chapter 8 addresses the design and management of data warehouses. Data warehouses include their own development tools, through which one can add new sources of data or provide new data cubes for new analysis techniques. The challenge is to provide a development environment that keeps track of the conceptual, logical, and physical perspectives of the data warehouse system and at the same time supports assessment of the quality of the data in the system. The solution to this challenge that is presented in the chapter consists of a set of nine interrelated modeling languages, one for each of the previously mentioned perspectives at each of three levels: data source, data warehouse, and client tool. A quality metamodel is placed on top of these nine languages. It allows quality goals and metrics to be expressed at any level and for any perspective. For example, the quality of the conceptual schema of a data source can be assessed as well as the performance of data uploaders in the physical perspective. The Telos models for this solution are also provided on the accompanying CD-ROM.

The last case study, in chapter 9, is about industrial design, in particular, the design of chemical processes. The goal of the Conceptual Lifecycle Process (CLiP) en-

[1] The CD-ROM resources are available online via http://conceptbase.cc//CB-Resources.html

vironment is to enhance chemical-plant design and shorten the time required for it. The core concept is a system that has properties that are organized into modeling aspects. Systems can be models of other systems (e.g., a system of equations can be a model of a valve). System properties are subdivided into property requirements, functions, realizations, behavior, and performance. These aspects cover the life cycle of the design of a technical system. The notion of a chemical process groups function, realization, behavior, and performance properties. In addition to these domain-specific concepts, a so-called workflow model is included that defines the system development method in CLiP. Telos metamodels of CLiP are provided on the accompanying CD-ROM. After validation of the metamodels with ConceptBase, the CLiP environment itself is realized on the basis of UML.

Theory versus Practice

The chapters in the first (theoretic) and second (practical) parts of the book are interrelated in various ways.

Goals

Chapters 1 and 2 elaborate on the necessity of a *goal perspective* in software development projects. Chapter 6 picks this up and presents a specific solution in the area of requirements engineering, in particular, for monitoring goals and maintaining a network of goal relations.

Social Aspects

Chapter 1 introduces *social aspects*: Humans take on roles and positions when discussing a system or an issue. This social aspect of software development projects recurs in several of the later chapters. Chapter 4 presents a method for organizing goals and requirements expressed by stakeholders and maintains the agreement level of the discussion between stakeholders. Chapter 9 devotes a separate metamodel to the social aspect in order to define collaboration in a team.

IRDS Abstraction Levels

The levels of the IRDS (chapters 2 and 3) are one of the organizing principles in this book. They distinguish a model from its definitions by a metamodel. All chapters in the second part of the book (i.e., chapters 4–9) are based on this principle.

Semantics

A particular problem is the formal semantics of abstraction principles like specialization. Chapter 3 presents metalevel formulas to facilitate the definition of semantics. Chapter 7 applies metalevel formulas to a new abstraction principle called materialization. Another interesting application is the formalization of the RDF language

that is so popular in the Semantic Web community (chapter 5). Here, it turns out that the RDF semantics can be expressed by adapting the Telos axioms discussed in chapter 3.

Model Analysis

The complexity and manifoldness of models require methods for assessing the models' correctness, completeness, and quality. Chapter 3 introduces queries as a facility for analyzing models in a formal way. Chapter 4 uses this idea to tackle the model analysis problem in the context of telecommunications modeling. Chapter 8 concentrates on the quantitative aspect of model analysis and presents a solution for managing measurements of models in the context of data warehouses.

Process Support

Systems are developed in a series of steps (chapter 2), some of which are about creating some results, and others of which are more informational. The case study presented in chapter 3 shows that process design can also be regarded as a modeling activity and that process enactment is the instantiation of the process model.

Multiple Perspectives

The complexity of software systems is addressed through specialized methods for the different aspects of a system yielding static, dynamic, and other models about the system (chapters 1 and 2). Chapter 8 employs an elaborate metamodel for data warehouse design with no less than nine interrelated aspects. Another example of representing multiple perspectives is presented in chapter 9 for the domain of chemical engineering.

The accompanying CD-ROM contains version 7.0 of the metadatabase system ConceptBase with executables for the platforms Windows and Linux. The examples for chapters 3, 7, 8, and 9 are provided on the CD-ROM to allow direct experiments with the metamodels presented in the book. The ConceptBase software tool and the practical examples included on the CD-ROM enable readers to start engineering their own methods directly on their computer.

Acknowledgments

First, we are grateful to the authors of the case studies documented in this book for their cooperation and patience. Their work has been partially published elsewhere but has never been collected in a single, integrated book. Their work offers strong evidence that metamodeling is a practical technique that can produce tangible results. Although the theoretic foundation is important for getting started with metamodeling and method engineering, we believe that only the examples in the extended

case studies can demonstrate the beauty and the feasibility of this exciting modeling technique.

This book would not have been possible without the work of our colleagues who have contributed to the design of Telos and implementation of the ConceptBase system over the last fifteen years. We are particularly grateful to Manolis Koubarakis and Alex Borgida for their contributions to the design of the Telos modeling language. Special thanks are also due to Martin Staudt, who developed the ConceptBase query language, and to Hans Nissen, who created its module system. Many thanks to René Soiron, Kai von Thadden, Rainer Gallersdörfer, and Thomas List for their great contribution to the ConceptBase server.

We thank the anonymous reviewers of an earlier draft of this book for their valuable suggestions and Michael Harrup, the careful copy editor, for improving the language and removing misspellings. We wholeheartedly appreciate the long-standing support of the MIT Press in general, and Douglas Sery, Valerie Geary, and Deborah Cantor-Adams in particular for their unwavering encouragement for this book. Last but not least, we thank Greg McArthur, who processed most of the copy editor's comments. His work was essential in finalizing this book.

Manfred A. Jeusfeld
Matthias Jarke
John Mylopoulos

Metamodeling for Method Engineering

1 A Sophisticate's Guide to Information Modeling

Alex Borgida and John Mylopoulos

slides

Models of various kinds of information about the world have found uses in diverse areas of computer science (e.g., artificial intelligence, databases, software requirements engineering), as well as in the business world (e.g., business process reengineering, corporate knowledge management). We provide a brief introduction to a variety of information modeling techniques by presenting a selective history, and then surveying a number of techniques for modeling static, dynamic, intentional and social aspects of an application. Our survey covers both diagrammatic and formal modeling methods, and applies them to an example involving scheduling meetings. Diagrammatic techniques, such as UML, are used because they visually summarize the principal elements of a model, and provide an easy to understand roadmap. Formal languages, such as KAOS, are based on predicate logic and capture additional details about an application in a precise manner. They also provide a foundation for *reasoning* with information models.

1.1 Introduction

Information modeling is concerned with the construction of computer-based symbol structures that model some part of the real world. We refer to such symbol structures as *information bases*, generalizing the term from related terms in computer science, such as *databases* and *knowledge bases*. Moreover, we refer to the part of a real world being modeled by an information base as its *application domain* (or just plain *application*). The atoms out of which one constructs the information base are assumed to denote particular individuals in the application (Maria, George, 7, ...) or concepts under which the individual descriptions are classified (student, employee, ...). Likewise, the associations within the information base denote real-world relationships, such as physical proximity and social interaction. We imagine that an information base is queried and updated through special-purpose languages, analogously to the way databases are accessed and updated through query and data manipulation languages.

It should be noted that in general, an information base is developed over a long time period, accumulating details about the application it models and changing to remain a faithful model of an evolving application. In this regard, it should be thought of as a *repository* that contains *accumulated*, *disseminated*, *structured* information, much like human long-term memory, or databases, knowledge bases, and so on. Assuming that information is entered into the information base through statements expressed in some language, the foregoing considerations suggest that the contents of these statements need to be extracted and organized. In other words, the organization of an information base should reflect its *contents*, not just its *history*.

This implies some form of a *locality principle* (Brodie 1984), which calls for information to be organized according to its subject matter. Support for such a principle may come from the tools provided for building and updating an information base, as well as from the development methodology adopted. For example, insertion operations that expect *object* descriptions (i.e., an object's name, attributes, superclasses, etc.) do encourage this type of grouping, in contrast to those that accept arbitrary statements about the application (e.g., an assertion like "Maria wants to play with the computer" or "George is outside").

What kinds of symbol structures does one use to build up an information base? Analogously to databases, these symbol structures need to adhere to the rules of some *information model*—a notion that is a direct adaptation of the concept of database data model. The following definition is also adapted from databases: An information model[1] consists of (1) a collection of symbol structure types, whose instances are used to describe an application; (2) a collection of operations that can be applied to any valid symbol structure, and (3) a collection of inherent constraints that define the set of consistent symbol structure states, or valid changes of states. The *relational model* for databases (Codd 1970) is the prototypical example of an information model. Its basic symbol structure types include tuple, table, and domain. Its associated operations include, for tuples, insert, delete, and update operations, and for tables, join and select operations. The relational model has a single inherent constraint: "No two tuples within a table can have the same key." Given the foregoing, one can define more precisely an information base as a symbol structure that is based on an information model and describes a particular application.

Is an information model the same thing as a *language* or a *notation*? For our purposes, it is not. An information model offers symbol structures for representing information. This information may be communicated to users of an information base (human or otherwise) through one or more languages. For example, there are several different query languages associated with the relational model, of which Structured Query Language (SQL) is the most widely used. In a similar spirit, we see notations as (usually graphical) partial descriptions of the contents of an information base.

Again, there may be several notations associated with the same information model (e.g., the different graphical notations used for data flow diagrams).

The earliest information models in computer science were *physical models*, which employed conventional programming notions (e.g., records, files, strings, and pointers) to build and maintain a data structure that modeled a particular application. Not surprisingly, such models focused mostly on *implementation*, as opposed to *representation*, aspects of the information being captured.[2] *Logical information models*, based on abstract mathematical symbol structures (e.g., sets, relations), were offered in order to hide implementation details from the user. The relational model for databases is an excellent example of a logical model. In a relational database, one does not need to know the physical data structures used (e.g., B-trees) in order to access the database's contents. Unfortunately, such models are not well-suited for modeling complex real-world applications. Finally, *conceptual models* offer facilities for modeling applications more "naturally and directly" (Hammer and McLeod 1981, 352), as well as for structuring and constructing information bases. These models provide *semantic terms* for modeling an application, such as "entity," "activity," "agent," and "goal," as well as means for organizing information in terms of *abstraction mechanisms*, which are often inspired by principles of cognitive science (Collins and Smith 1988).

Most of the conceptual models discussed in this chapter support some form of the locality principle alluded to earlier. There are several reasons for this. First, the principle appears to be consistent with accepted theories of human memory organization (Anderson and Bower 1973). Second, conceptual models supporting locality are generally considered more perspicuous, and hence easier to use. Finally, such models offer the promise of efficient implementations because of their commitment to clustering information according to its topic. In short, conceptual models adopting such a locality principle have advantages over standard methods, both on cognitive and on engineering grounds.

Information modeling touches on deep and long-standing philosophical issues, notably, the nature of generic terms included in an information base, such as `Person`, `Student`, and `Employee`. Do these terms represent abstract things in the application, in the same way `Michelle` or `Myrto` represent concrete ones? Or are these representations of concepts in the mind of the modeler? Philosophers as far back as Plato have taken stands on the problem. Plato, in particular, adopts a naive realism in which objective reality includes abstract ideas, such as the concepts of student and employee, and everything is out there to be discovered. Others, including Aristotle, Locke, and Hume, adopt various forms of conceptualism, according to which concepts are cognitive devices created through cognitive processes. For a discussion of the range of stands on this issue within philosophy and how these affect the nature of information modeling, see Artz 1997.

Likewise, information modeling touches on fundamental issues that relate to social science (Potts 1997). In particular, all the techniques discussed here adopt an *abstractionist* stance, founded on the notion of a model abstracted from an application, which captures the essence of the application, ignores bothersome details, and is intended for analysis and/or communication. Natural scientists and engineers use such abstractionist methods heavily. In contrast, *contextualism* emphasizes precisely the details and idiosyncrasies of each individual application, as well as the modeling process itself. These define a context and constitute the unique identity of each particular modeling situation. Ignoring them can lead to models that are inaccurate and misleading, as they simply miss the essence of each case. Contextualism has been largely developed and used within the social sciences, and it remains to be seen how one can combine it with abstractionism in information modeling.

The rest of the chapter is organized as follows. Section 1.2 presents a brief history of conceptual modeling in artificial intelligence, databases, software engineering, and information systems. Section 1.3 focuses on modeling the *static* aspects of an application dealing with scheduling meetings. The remaining sections present the modeling of *dynamic*, *intentional*, and *social* aspects for the same example. In each section we present both graphical and textual/formal models and discuss different kinds of analyses that can be carried out.

1.2 A Brief History

Over the years, there have been literally hundreds of proposals for conceptual models, most defined and used only within the confines of a single project. We review in this section some of the earliest models that launched fruitful lines of research and influenced the state of practice. Interestingly, these models were developed independently of one another and in different research areas within computer science.

Ross Quillian (1968) proposed *semantic networks*, a form of directed, labeled graph, as a convenient device for modeling the structure of human lexical memory. Nodes of his semantic network proposal (see figure 1.1) represented concepts (more precisely, word senses). For words with multiple meanings, such as "plant," there would be several nodes, one for each sense of the word (e.g., "plant" as in "industrial plant," "evergreen plant", etc.). Nodes were related through links representing semantic relationships, such as `isA` ("A bird is a(n) animal," "a shark is a fish"), `has` ("A bird has feathers"), and `eat` ("Sharks eat humans"). Moreover, each concept could have associated attributes, representing properties, such as "Penguins can't fly."

There are several novel ideas in Quillian's proposal. First, his information base was organized in terms of *concepts* and *associations*. Moreover, generic concepts were organized into an `isA` (or generalization) hierarchy, supported by attribute in-

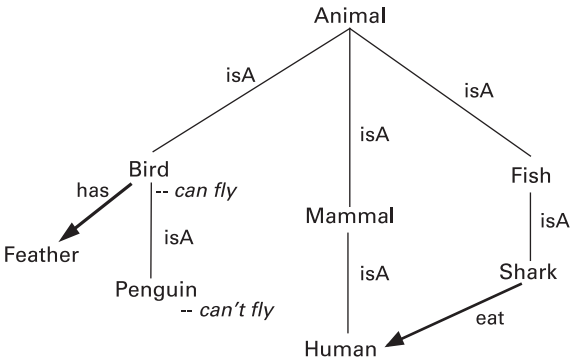


Figure 1.1
A simple semantic network

heritance. In addition, his proposal came with a radical computational model termed *spreading activation*. Thus, computation in the information base was carried out by "activating" two concepts and then iteratively spreading the activation to semantically adjacent concepts. For example, to discover the meaning of the term "horse food," spreading activation would fire the concepts `horse` and `food` and then spread activations to neighbors, until the following two semantic paths were discovered:

```
horse --isA--> animal --eats--> food
horse --isA--> animal --madeof--> meat --isA--> food
```

These paths correspond to two different interpretations of "horse food"; the first amounts to something like "food that horses eat," whereas the second refers to "food made out of horses."

In 1966, Ole-Johan Dahl and Kristen Nygaard proposed an extension of the programming language ALGOL 60, called *Simula*, intended for simulation applications. Simula (Dahl, Myrhaug, and Nygaard 1970) allows the definition of classes that serve as a cross between executable processes and record structures. A class can be instantiated any number of times. Each instance first executes the body of the class, to initialize it, and then remains as a passive data structure that can be operated upon only by procedures associated with the class. For example, the class `stack` in figure 1.2 has two local variables, `N`, an integer, and `T`, a vector of reals. Every time the class is instantiated, these are initialized. Then instances can be operated upon through two operations `push` and `pop` that perform (obvious) stack operations.

Simula advanced significantly the state of the art in programming languages, served as intellectual foundation for Smalltalk and has been credited with the launch of object-oriented programming. Equally importantly, Simula influenced information modeling by recognizing that for some programming tasks, such as simulation,

```
class stack (n); integer n;
 begin integer N; real array T[0;n];
   procedure push (Y); real Y;
       begin T(N) := Y; N := N + 1; end;
   procedure pop (Y); real Y;
       begin if N > 0 then
                 begin Y := T(N); N := N - 1; end;
             else /* print error message */ end;
  /* initialization follows */
 integer i;
 for i := 0 step 1 until n do T[i] := 0;
  N := 0;
end /* of stack class */
```

Figure 1.2
A Simula class definition

one needs to build a model of an application. For instance, to simulate a barbershop, one needs to define classes for the barbershop itself, its barbers, and customers, who arrive at a certain rate, wait in line, get a haircut by one of the barbers on hand, pay, and leave. According to Simula, such models are constructed out of class instances (*objects*, nowadays). These are the basic symbol structures that model elements of the application. Classes themselves define common features and common behaviors of instances and are organized into subclass hierarchies. Class declarations can be incorporated into subclasses through some form of inheritance (in this case textual).

Jean-Raymond Abrial proposed the *semantic binary model* for databases in 1974, shortly followed by Peter Chen's (1976) *entity-relationship model*.[3] Both were intended as advances over logical data models, such as Codd's relational model, proposed only a few years earlier.

The entity-relationship diagram of figure 1.3 shows entity types `Client`, `Book`, and `BookCopy` and relationships `requests` and `hasCopies`. Roughly speaking, the

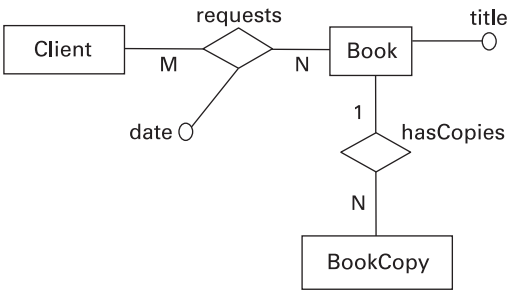


Figure 1.3
An entity-relationship diagram

diagram represents the fact that "Clients request books" and "Books have many copies." The `requests` relationship type is many-to-many (`N` to `M`), meaning that a client requests many books, and each book can be requested by many clients. On the other hand, `hasCopies` is a one-to-many relationship type in that a book can have many copies, but each copy is associated with a single book. In addition, values (integers, strings, etc.) can be associated with either entities or relationships through attributes (e.g., the string `title` of a book, the `date` when the request was made).

Novel features of the entity-relationship model include its built-in terms, which constitute *ontological assumptions* about the intended application. In other words, the entity-relationship model assumes that applications consist of *entities/values* and *relationships/attributes*. This means that the model is not appropriate for applications that violate these assumptions (e.g., a world of fluids, or those involving temporal events, state changes, and the like). In addition, Chen's original paper showed elegantly how one could map a schema based on his conceptual model, such as that shown in figure 1.3, to a logical database schema. These features made the entity-relationship model an early favorite, perhaps the first conceptual model to be used widely. Later research on semantic data models extended the basic ontology and constructs provided by Chen's proposal to facilitate the modeling of additional application semantics.

Abrial's semantic model is more akin to object-oriented data models, which became popular over a decade later, than Chen's entity-relationship model is. His model also offers entities and relations (albeit only binary ones) as primitive terms but includes a procedural component through which one can associate with a class four primitive operations: for adding instances of the class, deleting instances of the class, testing whether an object is an instance of the class, and fetching all class instances. These procedures can capture additional details of the situation being modeled.

Douglas Ross (1977; Ross and Schoman 1977) proposed in the mid-1970s the *Structured Analysis and Design Technique* (SADT) as a "language for communicating ideas" (Ross 1977, 17). The technique was used by Softech, a Boston-based software company, to specify requirements for software systems. According to SADT, the world consists of activities and data. Each activity is represented by a box and is described in part by the data involved in its execution: An activity may consume some data, represented through input arrows on the left side of the activity box, and produce some data, represented through output arrows on the right side, and may also have some data that control the execution of the activity but are neither consumed nor produced. (Control data are represented by arrows feeding into the activity box from the top.) For instance, the `Buy_Supplies` activity in figure 1.4 has input `Farm_Supplies`, output `Fertilizer` and `Seeds`, and controls `Prices` and `Plan&Budget`. An activity may be further described in terms of its own diagram, showing its subactivities. Thus `Grow_Vegetables` is defined in terms of the subactivities `Buy_Supplies`, `Cultivate`, `Pick_Produce`, and `Extract_Seeds`. An SADT model is therefore hierarchically structured, making it easier to build and understand than a nonhierarchical model. One of the more elegant aspects of the SADT conceptual model is its duality: Data, like activities, are also described in terms of diagrams

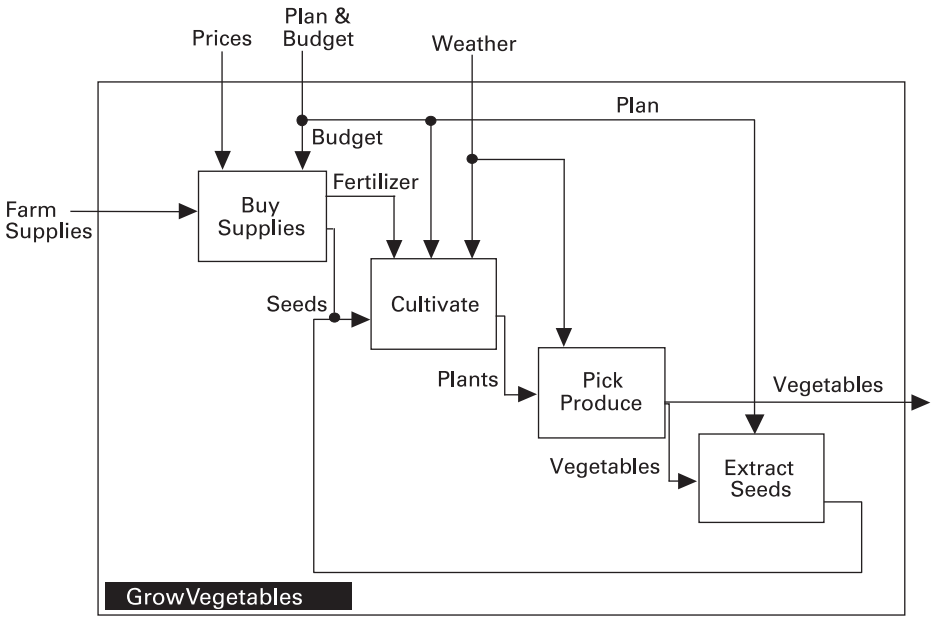


Figure 1.4
An SADT activity diagram

with input, output, and control arrows, which in this case represent activities that can produce, consume, or affect the state of a given datum.

Ross's contributions to information modeling include more advanced ontological assumptions: Unlike in the entity-relationship model, according to SADT, applications consist of both static and dynamic parts. Ross also was influential in convincing software engineers that it pays to have diagrammatic descriptions of how a software system is to fit its intended operational environment. This contribution helped launch requirements engineering as an accepted and important early phase in software development. The case for world modeling as part of requirements engineering was also articulated eloquently by Michael Jackson (1978), whose software development methodology (Jackson 1983) starts with a "model of the reality with which the system is concerned" (4).

The use of conceptual models for information systems engineering was launched by Solvberg (1979), and Bubenko's (1980) *conceptual information model* (CIM) is perhaps the first comprehensive proposal for a formal requirements modeling language. Its novel features include an ontology of entities and events and an assertional sublanguage for specifying constraints, including those expressing complex temporal relationships.

After these pioneers, research on conceptual models[4] and modeling broadened considerably, both in the number of researchers working on the topic and in the number of proposals for new conceptual models. In databases, dozens of new semantic data models were proposed, intended to "capture . . . more of the meaning of the data" (Codd 1979, 397). For instance, *RM/T* (Codd 1979) attempts to embed within the relational model the notion of entity and organizes relations into generalization hierarchies. *SDM* (Semantic Data Model) (Hammer and McLeod 1981) offers a set of highly sophisticated facilities for modeling entities and supports the organization of conceptual schemata in terms of generalization and aggregation, as well as a grouping mechanism. *Taxis* (Mylopoulos 1980) adopts ideas from semantic networks and Abrial's proposal to organize all components of an information system, including transactions, exceptions, and exception-handling procedures, using generalization hierarchies. Tsichritzis (1982) presents an early but thorough treatment of data models and modeling, and Hull and King (1987) and Peckham and Maryanski (1988) survey and compare a variety of semantic data models.

The rise of object orientation as the programming paradigm of the 1980s (and 1990s) led to object-oriented databases, which adopted some ideas from semantic data models and combined them with concepts from object-oriented programming (Zdonik and Maier 1989; Atkinson et al. 1990). Early object-oriented data models supported a variety of sophisticated modeling features (e.g., *Gemstone* [Copeland and Maier 1984], based on the information model of Smalltalk), but the trend in recent commercial object-oriented database systems is toward the information model

of popular object-oriented programming languages, such as C++. By including more and more programming aspects, object-oriented data models seem to be taking a step backward with respect to conceptual modeling.

The rise of the Internet and the World Wide Web has created tremendous demand for integrating heterogeneous information sources. This has led, in turn, to an emphasis on *metamodeling* techniques in databases, in which one needs to model the meaning and structure of the contents of different information sources, such as files, databases, Web sites, and digitized pictorial data, rather than an application (Klas and Sheth 1994; Widom 1995).

Within artificial intelligence (AI), semantic network proposals proliferated in the 1970s (Findler 1979), including those that treated semantic networks as a graph-theoretic notation for logical formulas. During the same period, Minsky (1975) introduced the notion of *frames* as a suitable symbol structure for representing commonsense knowledge, such as the concept of a room or of an elephant. A frame may contain information about the components of the concept being described and links to similar concepts, as well as procedural information on how the frame can be accessed and change over time. Moreover, frame representations focus specifically on capturing commonsense knowledge, a problem that still remains largely unresolved for knowledge representation research. Examples of early semantic network and frame-based conceptual models include *KL-ONE* (Brachman 1979) and *KRL* (Bobrow and Winograd 1977).

Terminologic/description logics are a family of formalisms that grew out of attempts to formalize and systematize semantic networks and frames. Such systems (e.g., CLASSIC [Borgida et al. 1989]) offer facilities for *precisely specifying* concepts in terms of necessary and/or sufficient conditions. For example, a bachelor might be defined as a person of male gender who does not have a spouse:

```
Bachelor == (and Person
                 (fills gender 'male)
                 (at-most 0 spouse))
```

The *description* on the right-hand side of the specification is built from identifiers of relationships (e.g., `spouse`), individuals (e.g., `'male`) and other concepts (e.g., `Person`), using concept constructors (**`and`**, **`fills`**, **`at-most`**) chosen from a small predefined set. The important point is that descriptions have a well-defined semantics and support *effective reasoning* (e.g., deciding when two descriptions are mutually inconsistent or one subsumes the other). In fact, from a theoretical point of view, description logics (DLs) such as KL-ONE and CLASSIC, have been the most thoroughly studied knowledge representation schemes. The decidability of reasoning in DLs is achieved by limiting what can be expressed in the language, with empirical research

driving the choice of particular concept constructors. A knowledge base management system using descriptions stores concept definitions (i.e., an ontology) in the "terminological" component; in addition, an "assertional" component is provided for stating which descriptions hold regarding specific individuals. Descriptions allow incomplete information to be encoded (e.g., we might know that Gianni has at least two children, who are older than sixteen, without knowing anything else about them) and used for reasoning. Borgida (1995) surveys the use of description logics in databases. Knowledge representation is thoroughly presented in Brachman and Levesque 1984, reviewed in Levesque 1986, and overviewed in Kramer and Mylopoulos 1991.

In requirements engineering, Sol Greenspan's RML (*Requirements Modeling Language*) (Greenspan, Mylopoulos, and Borgida 1982; Greenspan 1984; Greenspan, Borgida, and Mylopoulos 1986) attempts to formalize SADT using ideas from knowledge representation and semantic data models. The result is a formal requirements language in which entities and activities are organized into generalization hierarchies; Greenspan's proposal anticipates a number of object-oriented analysis techniques by several years. During the same period, the *GIST* specification language (Balzer 1981) was developed at the University of Southern California's Information Sciences Institute. It, too, was based on ideas from knowledge representation and supported modeling the environment; it was influenced by the notion of making the specification executable and by the desire to support transformational implementation. *ERAE* (Dubois et al. 1986) was an early effort that explicitly shared with RML the view that requirements modeling is a knowledge representation activity; it was founded on ideas from semantic networks and logic. The *KAOS* project constitutes a more recent and most significant research effort that strives to develop a comprehensive framework for requirements modeling and requirements acquisition methodologies (Dardenne, van Lamsweerde, and Fickas 1993). The language offered by KAOS for requirements modeling provides facilities for modeling goals, agents, alternatives, events, actions, existence modalities, agent responsibility, and other concepts. KAOS relies heavily on a metamodel to provide a self-descriptive and extensible modeling framework. In addition, KAOS offers an explicit methodology for constructing requirements that begins with the acquisition of goal structures and the identification of relevant concepts and ends with the definition of actions to be performed by the system to be built or agents existing in the system's environment.

The state of practice in requirements engineering was influenced by SADT and its successors. *Data flow diagrams* (e.g., De Marco 1979) adopt some of the concepts of SADT but focus on information flow within an organization, as opposed to SADT's all-inclusive modeling framework. The combined use of data flow and entity-relationship diagrams has led to an information system development methodology that dominated teaching and practice until the advent of the *Unified Modeling*

Language (UML). Since the late 1980s, however, object-oriented analysis techniques (for example, Shlaer and Mellor 1988; Rumbaugh et al. 1991; Jacobson et al. 1992) have been introduced in software engineering practice and are dominant today. These techniques offer a more coherent modeling framework than the combined use of data flow and entity-relationship diagrams. The object-oriented framework adopts features of object-oriented programming languages, semantic data models, and requirements languages. UML (Fowler and Scott 1997) integrates the features of these and other preceding object-oriented analysis techniques and has become the de facto system development standard for a considerable segment of the software industry.

An early survey of issues in requirements engineering appears in Roman 1985, and the requirements modeling terrain is surveyed in Webster 1987. Thayer and Dorfman 1990 includes a monumental tutorial on requirements engineering.

The history of conceptual modeling did not unfold independently within the areas reviewed here. An influential workshop held at Pingree Park, Colorado, in 1980 brought together researchers from databases, AI, programming languages, and software engineering to discuss conceptual modeling approaches and compare research directions and methodologies (Brodie and Zilles 1981). The workshop was followed by a series of other interdisciplinary workshops that reviewed the state of the art in information modeling and related areas (Brodie, Mylopoulos, and Schmidt 1984; Brodie and Mylopoulos 1986; Schmidt and Thanos 1989). The International Conferences on the Entity-Relationship Approach,[5] held annually since 1979, have marked progress in research on as well as the practice of conceptual modeling generally.

A number of papers and books survey the whole field of conceptual modeling or one or more of its constituent areas. Loucopoulos and Zicari 1992 is a fine collection of papers on conceptual modeling, most notably a survey of the field (Rolland and Cauvet 1992), and Boman et al. 1997 offers a complete and coherent approach to conceptual modeling, using Prolog as the representation and reasoning language. Mylopoulos and Brodie 1988 surveys the interface between AI and databases, much of it related to conceptual modeling. Along a similar path, Borgida 1990 discusses the similarities and differences between knowledge representation in AI and semantic data models in databases. It should also be acknowledged that the foregoing discussion has left out other areas in which conceptual modeling has been used for some time, most notably enterprise modeling (Vernadat 1996) and software process modeling (Madhavji and Penedo 1993).

In the remainder of the chapter, we consider several topics about what information may appear in a conceptual model: static and dynamic aspects, goals, agents, and intentions. In each case, we give a series of examples from the domain of meeting scheduling that illustrate issues that commonly arise in building such a model. The examples are expressed, whenever possible, in both a diagrammatic notation (such as UML) and a more formal, textual one (such as KAOS).[6]

1.3 Modeling Static Aspects of the Application

1.3.1 Individuals in the World

It is natural to see the world as being populated by *individuals*. Some are quite concrete, such as a particular person, Gianni, or a particular room in a particular building. Others are somewhat more abstract, like the meeting that Gianni attended last week, or the one he will be attending next week, or the Monday morning meeting he usually attends. These kinds of individuals have an intrinsic identity, so that even if we are told of two meetings to be held tomorrow morning at 9 A.M., we can distinguish them (e.g., count them), even if we cannot name any specific properties that they have and that are different in each. In order to refer to such individuals, it would be easiest if each had a unique name, but unfortunately the real world is never quite as neat as this. In this chapter, as in object-oriented systems, we will be assigning arbitrary identifiers to individuals (e.g., `gianni`), so that we can refer to them. Note that it is important to distinguish an individual from various *references* to it (`gianni` vs. "the person whose first name is 'John'" vs. "the initiator of tomorrow's meeting" vs. "the chairman of the psychology department").

Other individuals are mathematical abstractions, such as integers, strings, lists, and tuples, whose identity is determined by some procedure, usually involving the structure of the individual. We call such individuals *values*, in contradistinction to *objects*. For example, the two strings "abc" and "abc" are the same individual *value* because they have the same sequence of characters. Similarly for triangles with sides of length 25, 12, and 20 or dates such as 1925/12/20.

Although values are eternal, individuals usually have an associated period of existence. Time is therefore an intrinsic part of every object model, though frequently it is omitted, with the understanding that the model reflects only the state of the world *at the present moment*.

1.3.2 Classes

In general, building a model for an application begins with a model of the generic *concepts* that are relevant to the application. For example, if we are modeling a university, then concepts such as "faculty," "department," "degree," and "program of study" are modeled first, using the notion *class*. Like a database schema, these concepts serve to circumscribe the contents of the information base we are constructing. Some special individuals may show up at this stage as well, if they are sufficiently important and stable. For example, if we are modeling a pair of related universities (including, say, the University of North Bay), then the individual `northBayU` may, quite appropriately, appear in the model. In general, however, early modeling of an application focuses on classes of individuals, since usually there are too many individuals in the world to make modeling each of them realistic, and they come and

go, whereas the corresponding classes are more stable. When individuals are introduced in an information base, they are associated with one or more classes as their *instances*. So we distinguish a special `InstanceOf` relationship between individuals and classes.

If we are concentrating on meeting scheduling in the university world, some obvious classes of individuals are meetings (requested or scheduled), persons, committees, rooms, topics, dates, agendas, and timetables. Classes need to be given identifiers, and the choice of the names should be a matter of careful deliberation (Ross 1977a), because domain experts, who are most probably not computer experts, will rely on these to capture much of the semantics of the application. In UML, a class is presented as a box, labeled by the class identifier, as shown in figure 1.5.

1.3.3 Subclasses

For many classes, there are specialized subclasses, representing subconcepts that are also of interest. For example, a `Meeting` can be `DeptWide` or `UniversityWide`; it can also be `ScheduledRegularly` or `AdHoc`. Meetings might be classified depending on their intended purpose (hiring, curriculum, etc.), which often is correlated with a particular committee. In some of the above cases, we recognize that some subclasses are disjoint (e.g., `ScheduledRegularly` and `Adhoc`, or `DeptWide` and `UniversityWide`).

Note that a modeling language should not require mutual disjointness of subclasses. For example, we could have a meeting that discusses both hiring and curriculum and is therefore an instance of both the `Hiring` and `Curriculum` classes. It should be up to the modeler to decide what assertions make sense for her application,

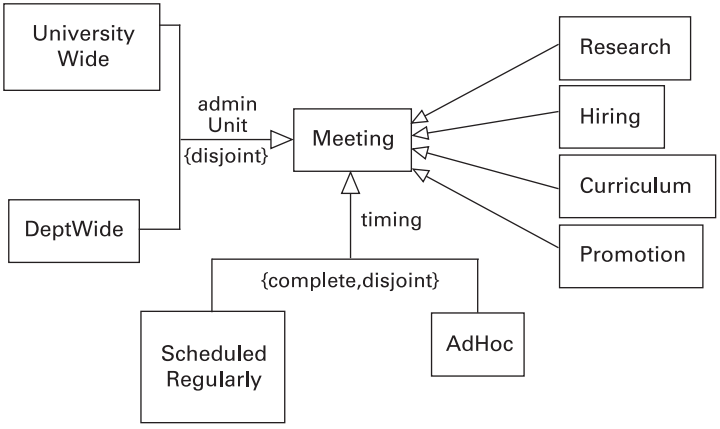


Figure 1.5
A subclass hierarchy for the class `Meeting`

so that they can be included in the specification of the information base. Unfortunately, in many modeling frameworks based on object-oriented programming languages, one is forced to create a common subclass for such situations, in order to guarantee a unique minimal class for every individual. Such features are not modeling principles—they are implementation obstacles.

Figure 1.5 illustrates the specification of subclasses in UML, using an open arrowhead. Grouping subclasses together under a single arrowhead allows one to specify that these subclasses are created based on some particular criterion (a discriminator attribute, such as `timing`) and that the subclasses are mutually disjoint (annotation {`disjoint`}) or that their union covers the entire superclass (annotation {`complete`}).

A final, conceptually important distinction concerning classes is whether an object's membership in a particular class can change with time or whether it is an intrinsic property. For example, it is fair to assume that a person remains a person throughout its lifetime. On the other hand, a faculty member is likely to start as a junior faculty member and become, in the normal course of events, a senior faculty member. (UML does not have special notation for this "dynamic" property.)

1.3.4 Modeling Relationships

Apart from being instances of classes, objects participate in relationships. Binary relationships are most frequent and are usually named directionally. For example, a meeting will have people participating in it (the `participants` relationship of a meeting), or conversely, a person will be participating in meetings (the `participates` relationship of persons). In addition, we may want to specify the minimum and maximum number of meeting participants (two or more, written as `2..*`) and the minimum and maximum cardinality of meetings a person can participate in (zero or more, written as `0..*`). This information is presented, according to the UML notation, as shown in figure 1.6.

Relationships can also exist between individuals and values. For example, a meeting may have a `time`, indicating the time period when it is to take place, and a `place`, describing the location. Frequently, such relationships are distinguished from relationships between individuals and are called *attributes*. In UML, attributes are shown schematically as named entries inside the box representing a particular class, as in figure 1.7.

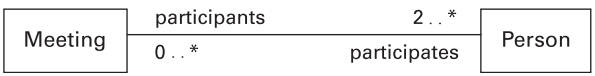


Figure 1.6
A semantic relationship between `Meeting` and `Person`

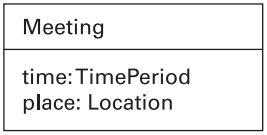


Figure 1.7
Attributes for `Meeting`

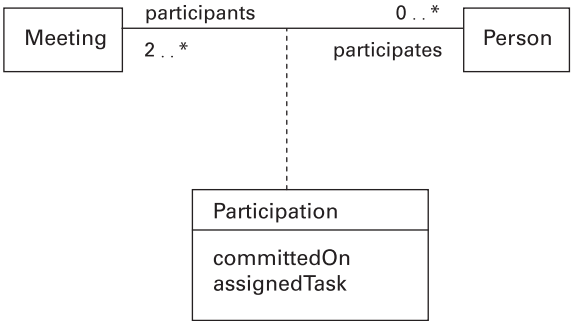


Figure 1.8
`Participation` as an association class

In addition to cardinality and range constraints on relationships, a wide variety of other kinds of constraints often need to be captured in a model. For example, because of the need for keys in relational databases, it is frequently desirable to specify that some collections of attributes uniquely identify an object within some class. And certain relationships can be designated to be *part-whole compositions* (marked with a solid diamond at the end of the composite), indicating among other things that when the whole is destroyed, its parts are also destroyed.

1.3.5 Reified Relationships

It is sometimes useful to attach attributes to qualify relationships. For example, when someone commits to attend a meeting, we might want to record when the commitment was made and also what task was assigned to the person for the meeting. In UML, this could be depicted using an association class, as shown in figure 1.8. Here the relationship shown in figure 1.6 has been augmented with two attributes, `committedOn` and `assignedTask`. These are attributes of each participation relationship, as opposed to attributes of the meeting and/or person involved.

We can reify relationships by treating them as individual objects, belonging to their own classes, as illustrated in figure 1.9. However, if `Participation` is just an ordinary class, one must be careful, because there may be multiple instances of the

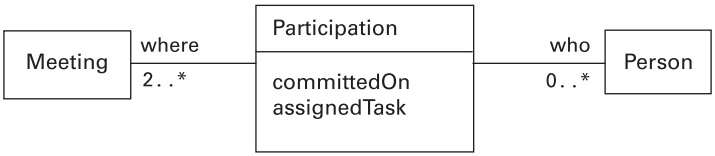


Figure 1.9
The reified `Participation` class

class with the same `where` and `who` attributes, which could not happen in an ordinary binary relationship (which is a set, rather than a bag, of tuples). One must therefore add constraints to ensure the uniqueness of the (`who`, `where`) pair of objects in the class. It is necessary to reify relationships when one is trying to represent *n*-ary relationships, such as room bookings, which relate a room, a time and a meeting. The entity-relationship model and KAOS (see section 1.3.7) are examples of conceptual models that treat arbitrary relationships as classes but distinguish them from entity classes.

1.3.6 A Larger UML Example

Figure 1.10 shows a UML class diagram for meeting scheduling. In addition to `Person` and `Meeting`, we include two subclasses to distinguish between requested and scheduled meetings. In addition, each person has a `Calendar`, which consists of constraints on meetings he or she can attend. (Since the deletion of a calendar does not imply the disappearance of meetings, this is not a composition relationship, and an open diamond is used to denote this "aggregation" relationship.) Meetings have associated requirements, involving equipment, space, location, and time, which are expressed as constraints.

1.3.7 A Formal Modeling Language

Like other formal conceptual models, KAOS allows one to express some of the preceding information through the use of built-in notions such as entity and relationship. For example, the UML material in figures 1.7 and 1.9, plus additional information about `Person`, is captured in the KAOS definition appearing in figure 1.11.

By associating a set with every class (the *extent* of the class), a function with every attribute, and a predicate with every relationship (as well as a predicate with every attribute), we obtain the basis of a first-order predicate language in which one can make assertions about the valid states of the world. The syntax illustrated in figure 1.11 is then provided a formal semantics by translation to formulas in predicate logic. For example, using the popular `x.f` notation as equivalent to `f(x)`, the `Meeting` entity class leads to the assertion of

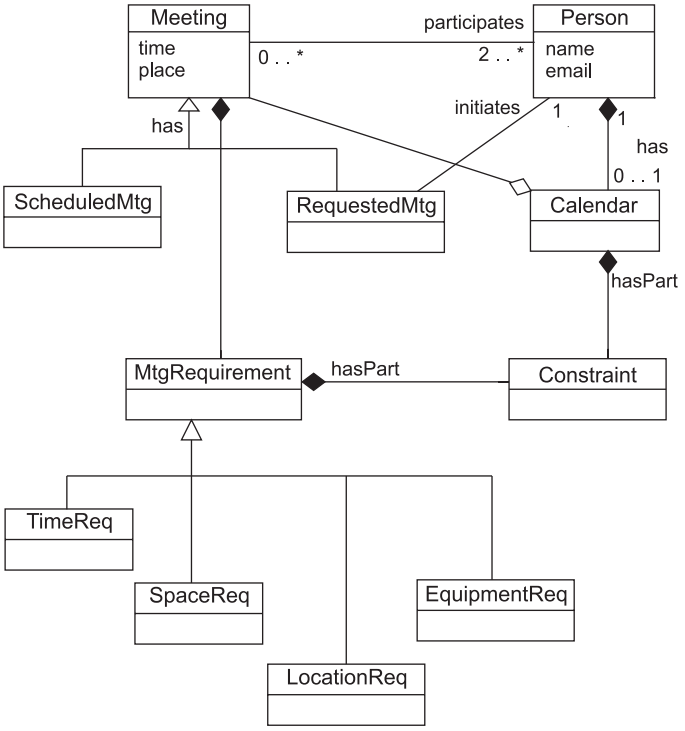


Figure 1.10
A UML class diagram for meetings

$(\forall$`m:MEETING)(TimePeriod(m.time)` $\wedge$ `RoomLocation(m.place)).`

Additional assertions, called *invariants*, can be added to the conceptual model to capture constraints for which there are no abbreviatory conventions. For example, we could require that the `busy` and `free` time intervals for a person must be disjoint,

$(\forall$`p:PERSON)(p.busy` $\cap$ `p.free =` $\emptyset$`),`

and also, that the times a person has committed to meetings should not be listed as free:

$(\forall$`p:PERSON,m:MEETING)(Participation(p,m)`
`=> m.time` $\notin$ `p.free ).`

1.3.8 Reifying Classes

Classes can themselves be treated as (meta)objects, with their own attributes and relationships. These are frequently useful for capturing aggregate information, which

```
entity Meeting
    has
         time : TimePeriod
         place : RoomLocation

entity Person
    has
         busy : SetOf (TimePeriod)
         free : SetOf (TimePeriod)

relationship Participation
    links Person {role participates, card 0:N}
          Meeting {role participant, card 2:N}
    has committedOn : Date
        taskAssigned : String
```

Figure 1.11
KAOS specification of `Meeting`

would not normally be associated with every individual in the class. For example, the class `Meeting` might have attributes that capture information such as the number of currently scheduled meetings or the average length.

1.3.9 Reasoning about the Static Model

Given the preceding translation into logic, it now becomes clear that we can perform logical inferences over the conceptual model. Most frequently, one is looking to discover *inconsistencies*, which can easily arise in large models. For example, the constraints imposed on an attribute in a class and one of its superclasses may be in conflict with each other. The result is that some classes or relationships can be proven to have no instances in any possible state of the world. Checks for consistency of this kind are often built into computer tools that support conceptual modeling with various languages and notations. Some of the most powerful logic-based reasoning about static models can currently be performed with notations that are translated into description logics, which include all manner of entity-relationship and object-oriented approaches, as discussed in Calvanese, Lenzerini, and Nardi 1998.

Figure 1.12
Some operations for `Meeting`

1.4 Modeling Dynamics

Of course, the world is not static, and therefore it is important to incorporate into an information model some of the dynamic aspects of the application being modeled. In the meeting-scheduling domain, some natural actions are issuing a meeting request, scheduling a meeting, and postponing or canceling a meeting. Other activities would involve the management of committees and their membership. In UML, activities are associated with specific objects as operations, shown in a separate compartment of a class diagram, after the attributes, as in the example in figure 1.12.

There are a number of aspects to modeling dynamics, supported by different notations.

1.4.1 Use Cases

An important technique for discovering what actions occur in some domain is describing *scenarios* involving actors and actions. Use cases (Jacobson et al. 1992) are specialized instances of such scenarios, describing at a very high level the interactions between a system to be and actors in its environment. As such, use cases offer an external view of an artifact (i.e., system object or property), and answer questions such as "What can the artifact do for the user?" and "How can the user use it?"

Assuming that the artifact is a meeting-scheduling system, we might start by identifying two types of actors, the (meeting) `Initiator`, who is unique for each meeting-scheduling case, and the (meeting) `Participants`, of whom there are two or more for each planned meeting. After a little thinking, we might identify six use cases for the problem at hand: `ScheduleMtg` (intended to initiate the scheduling process), `ProvideConstr` (in which participants describe their timetable for meetings), `GenSchedule` (which produces a schedule that takes into account a given set of constraints), `EditConstr` (modifying constraints previously submitted by a participant for a meeting), `Withdraw` (a participant from a meeting), and `ValidateUser` (for security purposes).

Let us elaborate on these use cases. First, the use case `ScheduleMtg` is triggered by the meeting initiator, who sends a message through the system to all intended participants requesting a meeting, provides them with his own constraints, such as his timetable and required equipment, and asks them for theirs. This use case launches the meeting-scheduling process supported by the system. The use case `Provide-Constr` is triggered by each participant when she is ready to fill in a form supplied by the system with the necessary meeting information. The system is supposed to inform the initiator that this step has been completed for each participant.

The use case `GenSchedule` generates schedules and is triggered by `ScheduleMtg` when any of the following events occurs:

- All possible participants have provided their constraints.
- A participant modifies her availability or withdraws from the meeting.
- The initiator modifies the meeting date range.

This use case produces a schedule if one is possible, given all participant constraints. Each time this step is completed, the system informs the initiator of the outcome. Likewise, the use case `EditConstr` is triggered by the initiator or a participant when he wants to change constraints he has previously submitted. The system is expected to send a message to the initiator after any amendments to the meeting constraints. The use case `Withdraw` is triggered by a meeting participant when she withdraws from the meeting. A withdrawal may or may not affect the time of the meeting. Again, the initiator is notified of any withdrawals. Finally, the use case `ValidateUser` is triggered by other use cases when a user attempts to log in. This use case is refined into the usual login protocol or a more elaborate one, depending on other requirements.

Actors, use cases, and the relationships between them are all modeled in UML with use case diagrams. A basic relationship between an actor and a use case is the `communication` association, shown as an unlabeled arrow in figure 1.13. This type of association can exist in one or both directions between an actor and a use case. A second semantic relationship between two use cases is labeled `uses` and indicates that one use case relies on another to realize its functionality. For example, `ValidateUser` is used by `ScheduleMtg`. A relationship of type `extends` is generally used to show optional or conditional behavior of a use case, which is carried out by another use case under certain conditions. `ProvideConstr`, for instance, extends the `EditConstr` use case by limiting its use to participants (the initiator provides her constraints in `ScheduleMtg`) and perhaps imposing additional restrictions on what users can change. The full use case diagram for a meeting-scheduling system is shown in figure 1.13.

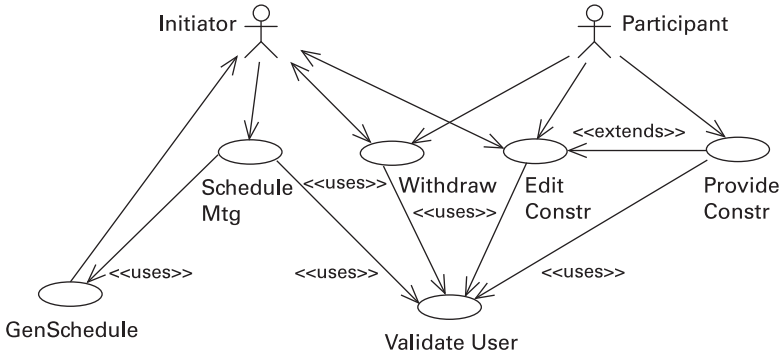


Figure 1.13
Use cases for a meeting-scheduling system

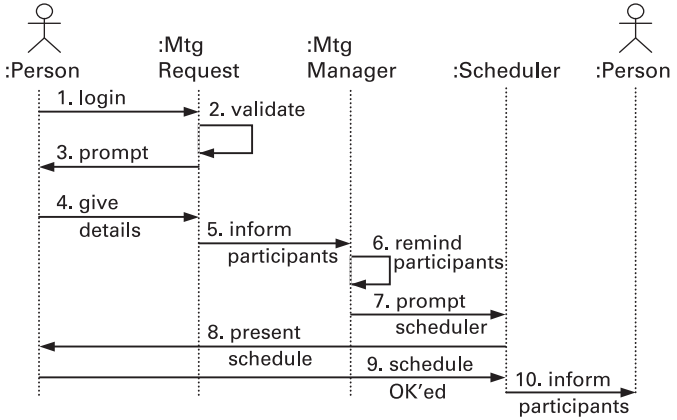


Figure 1.14
A sequence diagram for `ScheduleMtg`

1.4.2 Sequencing

Use cases offer a notation for describing activities at a very gross level, as (artifact) use. To provide additional details about activities, one can describe the *sequencing* of actions in various scenario instances, as well as the objects participating in the action.

Let us focus our example on `ScheduleMtg`, which launches the whole scheduling process. One scenario for this use case (expressed in figure 1.14 as a UML sequence diagram) begins with an initiator (a person) logging in, followed by the system validating her identity (say this is done by an instance of the class `:MtgRequest`), followed by a prompt to the initiator. (Notation: `:<Class>` refers to an unnamed instance of `<Class>`; e.g., the `:Person` associated with the first column of the se-

quence diagram in figure 1.14 refers to a particular person (the meeting initiator), whereas the last column refers to another person, a participant.) Then the initiator provides details about the meeting she wants scheduled, the system informs other participants, through the `:MtgManager`, which keeps prompting participants until they supply their meeting constraints. When all such constraints are in, the `:MtgManager` prompts another object, the `:Scheduler`, to generate a schedule. The schedule is relayed to the initiator for his approval and all participants are informed. Here `:MtgRequest`, `:MtgManager` and `:Scheduler` are respectively responsible for handling a meeting request, communication with participants, and scheduling. They could be actors, or components of a software system for meeting scheduling. Note that a use case may have several different scenaria, which follow different sequence paths. For instance, other paths may involve withdrawals on the part of some participants, revised constraints etc.

1.4.3 Formal Models of Dynamics

A description of an activity needs to characterize the transitions among the states with which it is associated. To complete such a description, one usually needs to (1) identify the participants in the activity (inputs, outputs, agents performing it or responsible for it, etc.) and (2) characterize the possible initial states in which the activity can be started and the final states in which it can end. Formal models of dynamic behavior augment graphical notations such as those discussed so far with a formal assertion language that includes primitives for talking about time.

Let's consider the activity `BookRoom` (specified in figure 1.15 using the KAOS language), which is performed by a person and involves directly a room, a time slot, and a meeting. In order for the activity to be carried out, the room must be free for the time slot chosen for booking. Once the activity is complete, the room is no longer free for that time slot, and a `Booking` relationship holds among the given parameters of the activity. Likewise, the `IssueReminder` activity takes as inputs a meeting and a person and produces a `Reminded` relationship. The postcondition of the activity says that when the activity has been completed, every participant has been reminded by the person in charge of the scheduling. `IssueReminder` also has a triggering condition: It is to be started if the meeting has been scheduled more than two weeks prior and there hasn't been a reminder in the previous week. These constraints are captured in figure 1.15.

One way to understand the figure's notation regarding temporal logic is to imagine that functions and relations have time as an extra argument. Then `Scheduled(m)` really means `Scheduled(m,now)`, while ■`Scheduled(m)` means that `Scheduled(m)` held at all time points before now: `(∀t:Time) before(t,now)=> Scheduled(m,t)`. The expression ■≤`2wk Scheduled(m)` constrains `t` to happen during the preceding two weeks:

```
operation BookRoom
  input Room{arg:r}, Meeting{arg:m}, Time{arg:t}
  output Booking
  PerformedBy MeetingManager{inst: mm}
  precondition t ∈ r.free
  postcondition t∉r.free ∧ t∈r.busy ∧ Booking(r,t,m,mm)

operation IssueReminder
  input Meeting{arg:m}, Person{arg:p}
  output Reminded
  postcondition
      (∀x:Person) Invited(x,m) ⟹ Reminded (p,x,m)
  triggercondition
      ■≤2wk Scheduled(m) ∧
      ¬( ◆≤ 1wk (∃r:"IssueReminder") Occurs(r) ∧ r.In=m )
  /* m was scheduled more than 2 weeks ago and there hasn't been a reminder
  within the last week */
```

Figure 1.15
Formal specification of `BookRoom` in KAOS

```
(∀t:Time) before(t,now)∧timeDist(now,t)≤2wk
              => Scheduled(m,t)
```

The notation ◆ provides an existential quantifier over past time. Note that in the formulas for the KAOS model in figure 1.15, the application of an operation is associated with the occurrence of an *event object* belonging to a class with the same name (but enclosed within quotation marks) plus an explicit predicate `Occurs`.

1.4.4 Complex Activities

UML distinguishes between actions and activities: The former are atomic, whereas the latter have duration, may overlap, and may have components which need to be coordinated. For example, scheduling of meetings involves

1. submitting a meeting request
2. obtaining participant calendar constraints, and concurrently
3. obtaining room availability constraints
4. evaluating the constraints to decide whether and how the meeting can be scheduled

In UML, the above sequence of steps can be described graphically using activity diagrams, which are inspired by process descriptions such as those found in the fields of workflow and software process modeling. In KAOS, operations can be combined into more complex scenarios using sequential, parallel, alternative, and repetitive composition.

Another formal notation for such complex activities is Congolog (DeGiacomo, Lespérance, and Levesque 1997), in which one starts with descriptions of atomic actions, such as `SubmitMeetingRequest`:

```
action SubmitMeetingRequest(init,mtg)
       possible when Person(init), Meeting(mtg)
       results in Requested(init,mtg) always;
```

and then describes composite actions using composition operators for sequencing (**;**), (nondeterministic) alternation (**||**), iteration, concurrent execution, (nondeterministic) choice, and so on:

```
activity ScheduleMtg(init,mtg,particips) =
  SubmitMeetingRequest(init,mtg);
  (  ObtainConstraints(particips, pcs)
    ||
     ObtainRoomConstraints(mtg.time, rcs) );
  EvaluateConstraints(pcs,rcs)
end activity
```

1.4.5 State Transition Diagrams

An object-centered alternative to describe behavior is to model the *life cycle* of an individual in terms of states and transitions induced by actions. For example, for meetings we may want to define the states a meeting can go through during its life cycle, say, `Scheduled`, `Cancelled`, and `Unscheduled`, and the allowable transitions among them. These transitions may be triggered by the occurrence of an action, such as `cancel` or `postpone`, or by the occurrence of events, such as the passing of a deadline. Figure 1.16 presents a reasonable state transition diagram for an already scheduled meeting.

Such state transition diagrams can be formalized in KAOS using the temporal operators introduced earlier. In particular, we could represent meeting states as predicates `Scheduled`, `Unscheduled`, and `Cancelled` and transitions as invariants on the entity `Meeting`. For instance, the transition from `Scheduled` to `Cancelled` might be specified by the following invariant:

```
(∀m:Meeting)Scheduled(m) ∧ (∃c:"cancel") Occur(c)
                    => ○Cancelled(m)
```

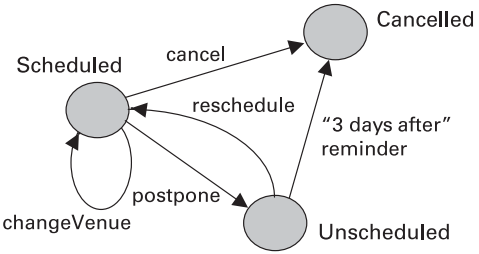


Figure 1.16
State transition diagram for meetings

The temporal operator ❍ declares its argument to be true at the next time instance. So the invariant depicted here declares that if a meeting `m` is currently in `Scheduled` state and the `Cancel` operation is applied, `m` will move to the `Cancelled` state. A fuller formal specification of `Meeting` might then include the material in figure 1.17.

1.4.6 Reasoning Using Dynamic Models

One can do a variety of type correctness checks on dynamic specifications, ensuring, for example, that definitions of operations and their uses have matching arguments. In addition, one can perform two kinds of computer-supported reasoning with formal descriptions of actions. The first is *enactment*: Given an initial description of some state of the world, one may be interested in finding out what can be determined about some state resulting from a particular sequence of operations. Congolog has exactly this capability, because Congolog descriptions can be translated into Prolog programs. A second kind of reasoning that may be conducted regarding a set of formal activity specifications is determining whether the total system obeys some theorem/invariant such as termination or lack of deadlock. This is usually accomplished by humans through the use of special theorem-proving aids (Lespérance et al. 1999), unless the logic used to describe actions is decidable.

1.5 Modeling Goals and Intentions

A third modeling dimension of any application encompasses the world of things agents believe in, want, prove or disprove, and argue about. This dimension covers concepts such as "issue" and "goal" and relationships such as "supports," "denies," and "subgoalOf." The subject of beliefs and goals has been studied extensively in AI. For instance, Maida and Shapiro 1982 addresses the problem of representing propositional attitudes, such as beliefs, desires, and intentions, for agents. As shown in requirements modeling research, such as Feather 1987 and Dardenne, van Lamsweerde, and Fickas 1993, the modeling of goals is an important component of soft-

```
entity Meeting

 has

     time: TimePeriod

     place: RoomLocation

invariant

  (∀m:Meeting)Scheduled(m) ∧(∃c:"cancel")Occur(c) ∧

     c.In=m ⟹  ❍Cancelled(m)

  (∀m:Meeting)Scheduled(m) ∧ (∃p:"postpone")Occur(p) ∧

     c.In=m ⟹  ❍Unscheduled(m)

  (∀m:Meeting)Scheduled(m) ∧ (∃c:"changeVenue")Occur(c) ∧

      c.In=m ⟹  ❍Scheduled(m)

  (∀m:Meeting)Unscheduled(m) ∧(∃c:"reschedule")Occur(c) ∧

      c.In=m ⟹  ❍ Scheduled(m)

  (∀m:Meeting) ■ ≤ 3days Unscheduled(m)⟹  ❍ Cancelled(m)
```

Figure 1.17
KAOS specification of `Meeting` including state transitions

ware specification and design. And of course, goals are of primary importance in the analysis of enterprises.

1.5.1 Modeling Issues

Modeling the issues that arise during complex decision making is discussed in Conklin and Begeman 1988. The application of such an argumentation framework to software design, intended to capture the arguments pro and con regarding decision problems and the decisions they result in, has been a fruitful research direction since it was first proposed in Potts and Bruns 1988, with notable refinements described in MacLean et al. 1991 and Lee 1991. For example, MacLean et al. 1991 models design rationale in terms of *questions*, *options*, and *criteria*. In designing an automated teller

machine (ATM), for instance, the designer may want to ask questions such as "What range of services will be offered (by the ATM under design)?" There may be two options: full range and cash disbursement only. In turn, there may be two criteria for choosing among them: user convenience and cost. On a complementary front, Gotel and Finkelstein 1995 studies the types of contributions a stakeholder can make to an argumentation structure.

1.5.2 Goals

Going as far back as the late 1960s, AI planning and problem solving used AND/OR trees as basic data structures to define and explore alternative ways of satisfying a goal. Briefly, a particular goal to be satisfied was decomposed iteratively, using AND and OR, until (sub)goals were encountered that were either trivially satisfied ("solved") or unsatisfiable ("unsolvable"). A goal could be, for example, a desired world situation, expressed as a logical formula; then the task for a planning program would be to find a sequence of actions that could lead to the desired situation.

We could consider the meeting-scheduling task as a generic goal to be achieved and then use an AND/OR decomposition to explore alternative solutions. Each alternative would be a potential plan for satisfying the goal. Figure 1.18 presents such a decomposition of the `ScheduleMtg` goal. AND decompositions are marked with an arc, indicating that satisfying the goal that appears above the arc can be accomplished by satisfying all the subgoals encompassed by the arc; OR decompositions, on the other hand, are marked with a double arc (or "new moon" symbol) and

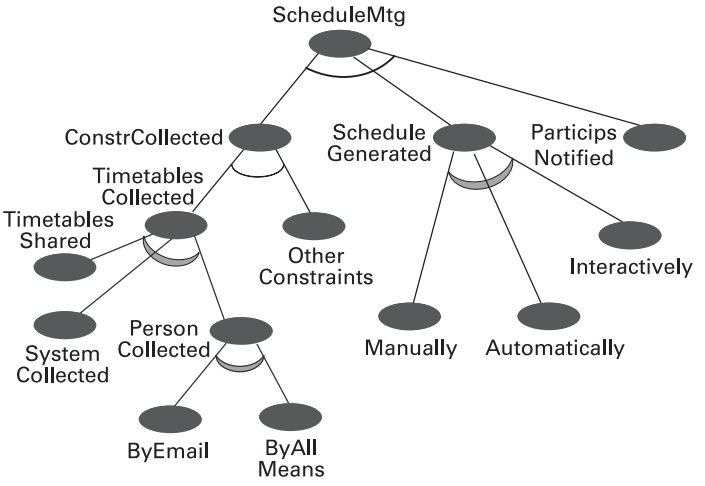


Figure 1.18
A (partial) space of alternatives for meeting scheduling

require only that one of the alternate subgoals encompassed by the double arc be satisfied. For our example, the goal `ScheduleMtg` is first AND-decomposed into subgoals `ConstrCollected`, `ScheduleGenerated`, and `ParticipsNotified`, all of which must be satisfied for the meeting to be scheduled. In turn, `ScheduleGenerated` is OR-decomposed into three subgoals, which find, respectively, a schedule manually, automatically (e.g., schedules are retrieved from a database), or interactively; any of these three methods of determining a schedule can satisfy the subgoal of schedule generation. Other decompositions explore alternative ways of fetching the necessary information, including timetable information, which may or may not be publicly available for each potential participant. Alternatives also consider whether only the initiator or all participants can specify other constraints prior to scheduling. As the reader may well appreciate, even with this simple example, there could be literally dozens of alternative solutions, and the solution sketched in the previous section is but one of them.

KAOS offers a formal language for describing goals, together with an ontology for classifying goals and indicating their decomposition. Thus, the top of the goal tree in figure 1.18 would be described as

```
  SystemGoal Achieve[ScheduleMtg]
   InstanceOf SatisfactionGoal
   ReducedTo ConstrCollected, ScheduleGenerated, ParticipsNotified
   FormalDef (∀m:Meeting, init:Initiator)
Requested(m,init)∧Feasible(m) => (◇ ≤ 3day Scheduled(m))
```

where, for example, the keyword *Achieve* describes a pattern of temporal behavior in which some target condition must eventually be established by the agent to whom the goal is assigned (in this case, the system). The advantage of goal classification is the availability of a knowledge base of *generic* ways to elaborate or achieve goals, which forms the basis of tools for supporting requirements acquisition (van Lamsweerde, Darimont, and Massonet 1995). The **FormalDef** part of the description shows a possible formal specification of the goal in terms of some of the predicates used in the description of actions—in this case requiring that the meeting be scheduled no more than three days from the time it was requested.

1.5.3 Reasoning with Goals

Goal formalizations, such as the ones in the foregoing, can be used for a variety of tasks, including detecting and resolving conflicts among goals, revealing high-level exceptions that may obstruct the accomplishment of goals, and proving that a goal decomposition is correct and complete or that a set of operations ensures the goals it operationalizes (e.g., Dardenne, van Lamsweerde, and Fickas 1993; Darimont and van Lamsweerde 1996).

1.5.4 Softgoals

The preceding notions are helpful when goals can be crisply specified, such as that of wanting to have a meeting scheduled. For software systems, one often also needs to describe so-called *nonfunctional requirements*, or *qualities*, such as "system must be usable" or "system must improve meeting quality."

Some nonfunctional goals (e.g., those dealing with safety or security) can be treated in the manner described in the foregoing, but others don't have generic definitions, nor do they have clear-cut criteria for establishing when they have been satisfied. For these, we need a looser notion of goal and a richer set of relationships so that we can indicate, for example, that a goal supports or hinders the accomplishment of another one.

To model this looser notion of goal, we use the notion of *softgoal*, proposed by the nonfunctional requirements (NFR) framework of Lawrence Chung (Chung et al. 1999). Softgoals are concepts intended to represent precisely such ill-defined goals and their interdependencies. To distinguish them from their (hard) goal cousins, we will say that a softgoal is *satisficed* when there is sufficient positive evidence for it and little negative evidence against it, and it is *unsatisficable* when there is sufficient negative evidence against it and little positive evidence for it.

Let's give an example concerning the quality "highly usable system," which may be as important an objective for the system-to-be as any of the functional goals encountered earlier. The softgoal `Usability` represents this requirement in figure 1.19.[7] The process for analyzing it, as with goals, consists of iterative decompositions, which involve similar AND/OR relationships or other, more loosely defined dependency relations. In the figure, the arrows between a number of softgoals describing such relationships are labeled with a plus sign, which indicates that the softgoal at which the arrow begins supports (or "positively influences") the softgoal at which the arrow terminates. For instance, `UserFlexibility` is clearly enhanced by the system quality `Modularity`, which allows for substitutions of modules, and also by the system's ability to allow setting changes (`AllowChangeOfSettings`). These factors, however, are not claimed to be necessarily sufficient to satisfice `User-Flexibility`; hence the relationships are marked with plus signs instead of the arcs that would indicate AND/OR relationships.

Figure 1.19 gives only a partial decomposition of the softgoal `Usability`. The softgoals `ErrorAvoidance`, `InformationSharing`, and `EaseOfLearning` have their own rich space of alternatives, which may be elaborated through further refinements.

For any given software development project, several softgoals will have been set down initially as required qualities of the software developed. Some of these may be technical, such as (system) `Performance`, because they refer specifically to qualities of the system to be. Others will be more business-oriented. For instance, it is reason-

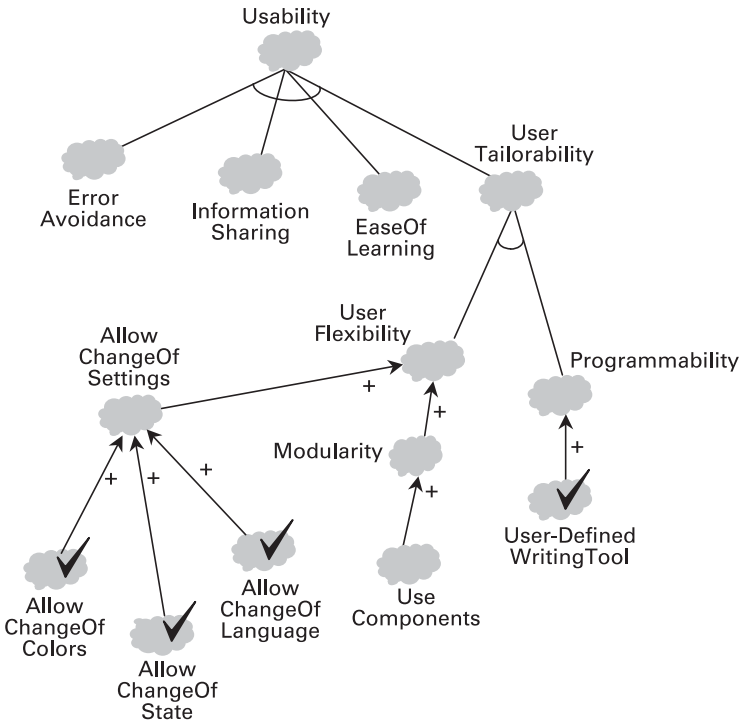


Figure 1.19
A (partial) softgoal hierarchy for `Usability`

able for a firm's management to require that the introduction of a new meeting-scheduling system improve meeting quality (by increasing average participation and/or effectiveness measured in some way) or cut average cost per meeting (where costs include those incurred during the scheduling process). Softgoal analysis calls for each of these qualities, represented as softgoals, to be analyzed in terms of a softgoal hierarchy, such as the one shown in figure 1.19.

1.5.5 Softgoal Correlations

The softgoal hierarchies discussed in the previous section are built by repeatedly asking the question "What can be done to satisfice or otherwise support this softgoal?" Unfortunately, softgoals are frequently in conflict with one another. Consider, for instance, security and user friendliness, performance and flexibility, and high quality and low costs. Correlation analysis is intended to discover positive or negative lateral relationships between softgoals. Such analysis can begin by noting top-level lateral relationships, such as, say, a negatively labeled relationship[8] between `Performance`

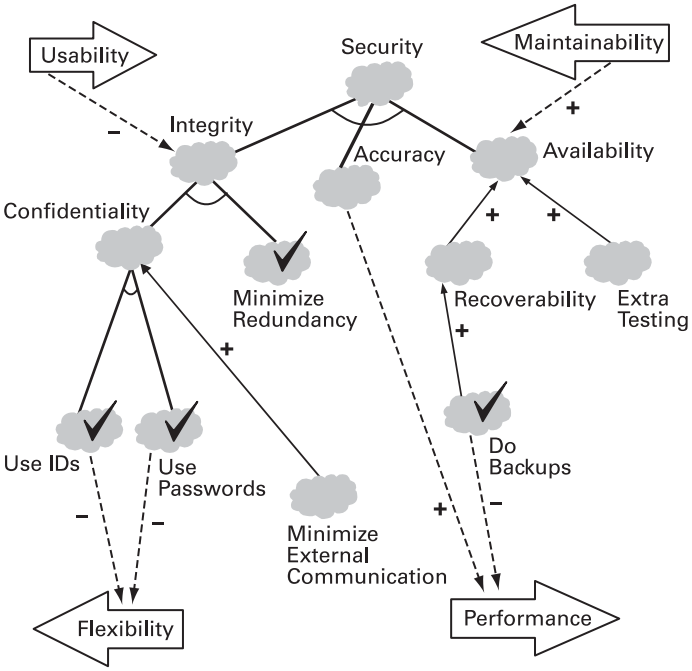


Figure 1.20
A (partial) softgoal hierarchy for `Security`, including correlations with other hierarchies

and `Flexibility`. This relationship can then be refined to one or more relationships of the same type from subgoals of `Performance` (say, `Capacity` or `Speed`) to subgoals of `Flexibility` (say, `Programmability` or `InformationSharing`). This process is repeated until the point is reached at which relationships cannot be refined farther down the softgoal hierarchies. Figure 1.20 shows diagrammatically the softgoal hierarchy for `Security`, with correlation relationships to other hierarchies.

1.5.6 Comparing Solutions

Suppose now that we'd like to compare alternative solutions to the goals in figure 1.18 in regard to all the softgoals identified so far, since we propose to use the latter in order to evaluate the former. For example, alternative subgoals of the goal `ScheduleMtg` will require different amounts of effort for scheduling. With respect to these softgoals, automation is desirable, whereas doing things manually is not. On that basis, we can set up positively or negatively labeled relationships linked to subgoals such as (`Schedule Generated`) `Automatically` or `Manually` (shown in figure 1.21). On the other hand, if meeting quality is the criterion, scheduling the meeting manually is actually desirable (because, presumably, it adds a personal

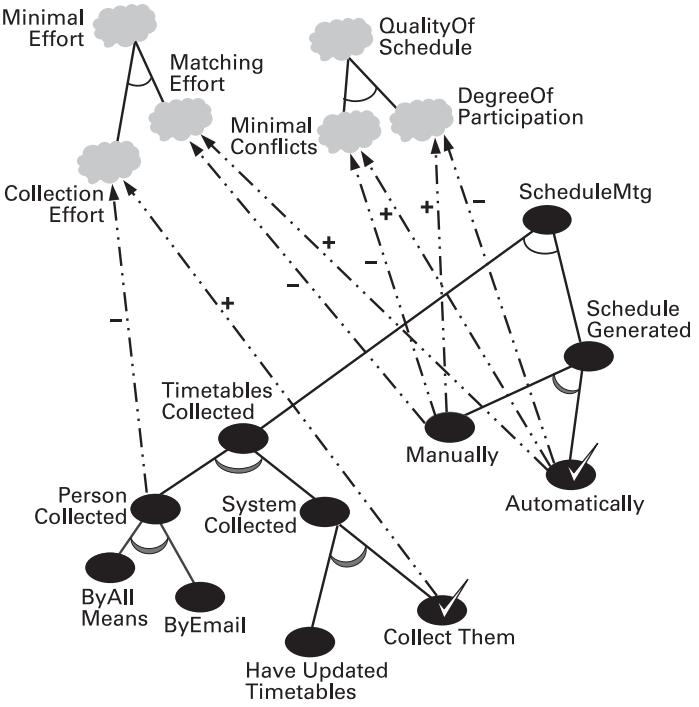


Figure 1.21
Evaluation of the goal `ScheduleMtg`

touch to the scheduling process), whereas doing things through the system receives low marks.

Figure 1.21 shows a possible set of correlation links for a simplified version of the `ScheduleMtg` goal in terms of the softgoals `MinimalEffort` and `QualityOf-Schedule`. The goal tree structure in the center right of the figure shows the refinement of the `ScheduleMtg` goal, whereas the two softgoal trees in the upper part of the figure represent the softgoals that are intended to serve as evaluation criteria.

This example points out a major advantage of treating softgoals (such as `Usability`) as goals in their own right, rather than as qualifiers of other goals (e.g., `UsableSystem`): It encourages the separation of analysis of a quality (e.g., `Usability`) from the object to which it is applied (e.g., `System`) and from other attributes. This allows relevant knowledge to be brought to bear on the analysis process: from very generic (e.g., "To achieve quality *x* for an artifact, try to achieve *x* for all of its components") to very specific (e.g., "To achieve effectiveness of a software review meeting, all stakeholders must be present"). Knowledge-structuring mechanisms such as classification, generalization, and aggregation can be used to organize the

available know-how to support such a goal-oriented analysis process. A thorough account of such generic as well as specific knowledge for softgoals such as `Security`, `Accuracy`, and `Performance` can be found in Chung et al. 1999, along with case studies that evaluate the effectiveness of the nonfunctional requirements framework.

1.6 Modeling Social Settings

The fourth modeling dimension we'll consider here covers social settings, including permanent organizational structures, group collaborations, and shifting networks of alliances and interdependencies (Galbraith 1973; Mintzberg 1979; Pfeffer and Salancik 1978; Scott 1987). Traditionally, this dimension has been characterized in terms of concepts such as `actor`, `position`, `role`, `authority`, and `commitment`. Yu (1993; Yu and Mylopoulos 1994; Yu, Mylopoulos, and Lespérance 1996) proposes a strategic dependency model that includes a novel set of concepts for modeling organizations.

According to this model, an organization is described in terms of *actors* and *dependencies*. Actors can be `agents` (such as Michelle), `positions` (e.g., company president), or `roles` (e.g., the chair of a meeting). Actors can be related to one another through links that represent (social) dependencies. Each dependency between two actors indicates that one actor depends on the other for something in order to attain some goal. We call the depending actor the *depender* and the actor who is depended upon the *dependee*. For example, a professor (the depender) can achieve the goal of scheduling a meeting of the tutors for her course by depending on her secretary (the dependee). Without the opportunity of using the services of the secretary, the professor may not be able to achieve the goal (for lack of time or lack of information about the tutors' e-mail addresses). On the other hand, the professor is vulnerable to the secretary's forgetting her request to schedule the meeting or otherwise not doing a proper job of it. The model distinguishes among four types of dependencies (goal, task, resource, and softgoal dependency) based on the type of freedom that is accorded to the dependee in fulfilling its obligation to the depender. In addition, three levels of dependency strengths are distinguished, based on the degree of vulnerability due to the effects of changes to dependent goals. For instance, assume that the `ScheduleMtg` goal is associated with anyone who can play the role of (meeting) initiator and that professors can play such a role. Then clearly, there are different ways to satisfy this goal, such as by the initiator's delegating the meeting-scheduling task to someone else (e.g., her secretary), or by her keeping the responsibility, but simply delegating some of the component tasks. In the latter case, she might delegate the task of keeping people's calendars updated or of finding a meeting time for a given a set of scheduling constraints and preferences.

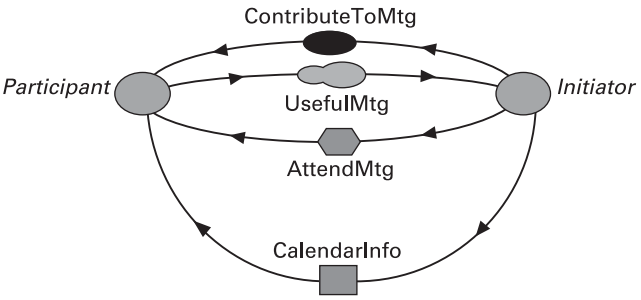


Figure 1.22
Initiator-participant dependencies for a meeting

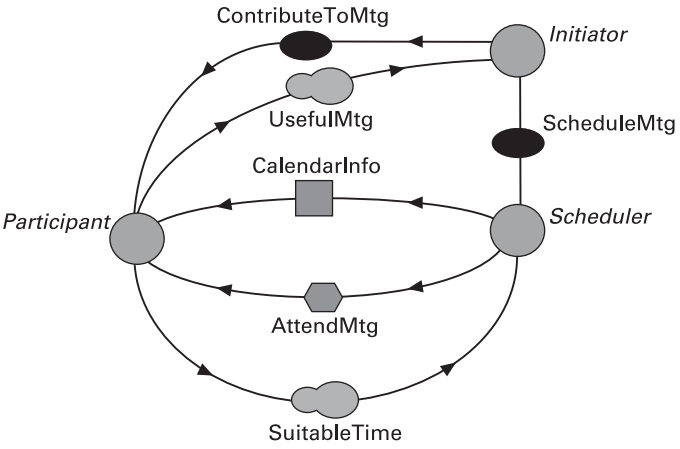


Figure 1.23
Initiator relies on scheduler to schedule meeting

Figure 1.22 shows a partial set of dependencies between an initiator and a participant, assuming that the initiator collects calendar information for each participant and does the scheduling herself. Here the initiator depends on each participant to provide calendar information (bottom of the figure). This is a resource dependency. The initiator also depends on each participant to attend the meeting (a task dependency) and contribute to it. The latter is clearly a goal dependency, because the initiator obviously doesn't care *how* this is done, as long as it is done. In turn, each participant depends on the initiator to organize a useful meeting. This is an example of a softgoal dependency, since "useful meeting" is not a well-defined goal.

Figure 1.23 shows an alternate structure of strategic dependencies in which the initiator relies on a scheduler to schedule the meeting. The scheduler's task is to gather

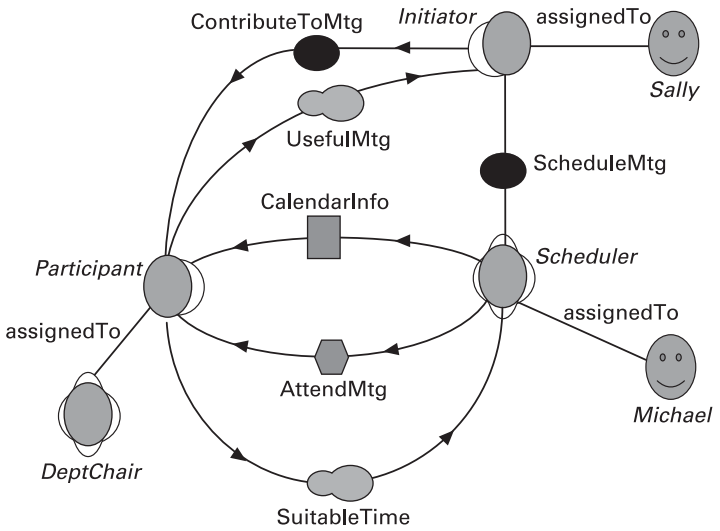


Figure 1.24
Agents, positions, and roles

calendar information from participants and have them show up at the meeting. Participants, in turn, depend on the scheduler to choose a suitable time. The initiator still depends on participants to contribute to the meeting, and they on him to ensure that the meeting is useful.

As indicated earlier, actors can be agents, positions, or roles. Figure 1.24 elaborates on the distinction by showing the `Initiator` and `Participant` actors as roles, whereas the `Scheduler` actor is labeled as a position. We also show two agents (`Sally` and `Michael`) who are assigned to the `Initiator` role and `Scheduler` position, respectively; the `DeptChair` (a position) has been assigned to the `Participant` role. Note that the figure doesn't show certain kinds of information, such as how many assignments can be associated with each position and/or role.

1.6.1 Strategic Rationale Analysis

To explore and analyze how alternative dependency configurations fare with respect to a set of qualities, an analyst may attempt to explicitly model and analyze the rationales behind the alternatives. Continuing with the meeting-scheduling example, let us say there are several groups within the organization having the same meeting-scheduling problem. One could let each group solve the problem individually or offer a centralized solution in which certain meetings are organized by a central scheduler, to ensure effectiveness and spread the remaining meetings. Alternatively, there might be some organization-wide facilitation, such as a centralized calendar database,

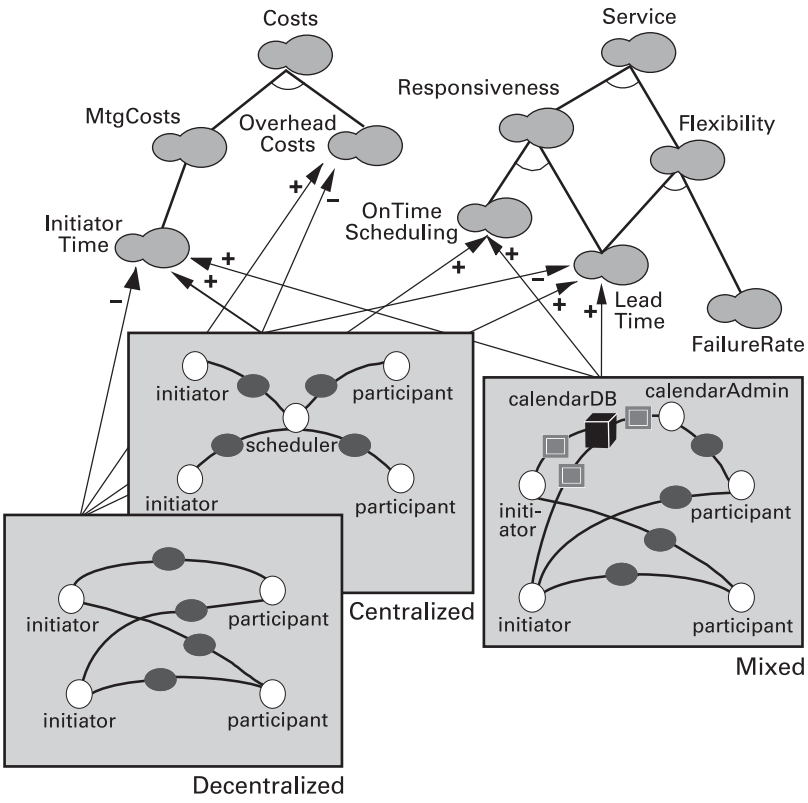


Figure 1.25
Evaluating alternative strategic dependency configurations

administered by a specific person. How does one go about comparing these alternatives? Qualities represented as softgoals can be used for this type of comparison.

Figure 1.25 assumes that the scheduling-process alternatives are to be evaluated with respect to overall `Costs` and quality of the `Service` provided. These two qualities are further refined into several other softgoals. The three alternative meeting-scheduling designs (centralized, decentralized, and mixed) can now be correlated with several of the softgoals, as shown in the figure. These correlations can serve as rationale for choosing a scheduling process within a large organization.

1.7 Summary

We have reviewed basic modeling techniques from different areas of computer science and have briefly introduced concepts that can be used to model the static,

dynamic, intentional, and social aspects of an application. We hope that this quick tour of information modeling and conceptual models has helped the reader appreciate the breadth and depth of the subject matter, as well as its central role for computer and information science.

Acknowledgments

We are extremely grateful to Axel van Lamsweerde for his detailed and insightful comments on an earlier draft of the chapter. All remaining errors are of course our own responsibility.

The work of Alex Borgida was funded in part by the U.S. National Science Foundation under grant no. IRI-9619979. John Mylopoulos was funded partly by Communications and Information Technology Ontario (CITO) and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Notes

1. Adapted from Ted Codd's (1982) classic account of data models and databases.

2. It is interesting to note that the Y2K problem was caused precisely by this tension between implementation and representation concerns.

3. The model was actually first presented at the First Very Large Databases Conference in 1975.

4. The term "conceptual modeling" was used in the 1970s either as a synonym for semantic data modeling or in the technical sense of the ANSI/X3/SPARC report (ANSI/X3/SPARC Study Group 1975), in which it refers to a model that allows the definition of schemata lying between external views, defined for different user groups, and internal ones defining one or several physical databases. The term was used more or less in the sense discussed here at the Pingree Park workshop "Data Abstraction, Databases and Conceptual Modeling," held in June 1980 (Brodie and Zilles 1981).

5. Eventually renamed the International Conferences on Conceptual Modeling.

6. For completeness, we note that KAOS has its own graphical notation, based on semantic networks.

7. Figure 1.19 was adapted from work prepared by Lisa Gibbons and Jennifer Spiess for a graduate course taught by Eric Yu during the spring term of 1996.

8. A negatively labeled relationship indicates that the fulfillment of one goal negatively influences the fulfillment of the other goal.

References

Abrial, J.-R. 1974. "Data Semantics." In *Data Base Management, Proceedings of the IFIP Working on Conference Data Base Management*, ed. J. W. Klimbie and K. L. Koffeman, 1–60. Amsterdam: North-Holland.

Anderson, J., and G. Bower. 1973. *Human Associative Memory*. Washington, DC: Winston-Wiley.

ANSI/X3/SPARC Study Group on Database Management Systems 75-02-08. 1975. "Interim Report." *FDT* FDT-ACM SIGMOD Record 7, no. 2: 1–140.

Artz, J. 1997. "A Crash Course on Metaphysics for the Database Designer." *Journal of Database Management* 8, no. 4: 25–30.

Atkinson, M., F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. 1990. "The Object-Oriented Database System Manifesto." In *Deductive and Object-Oriented Databases, Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89)*, ed. W. Kim, J.-M. Nicolas, and S. Nishio, 223–240. Amsterdam: Elsevier Science.

Balzer, R. 1981. "Final Report on GIST." Technical report, Information Sciences Institute, University of Southern California, Marina del Rey.

Bobrow, D., and T. Winograd. 1977. "An Overview of KRL, a Knowledge Representation Language." *Cognitive Science* 1: 3–46.

Boman, M., J. Bubenko, P. Johannesson, and B. Wangler. 1997. *Conceptual Modeling*. Upper Saddle River, NJ: Prentice Hall.

Borgida, A. 1990. "Knowledge Representation, Semantic Data Modeling: What's the Difference?" In *Proceedings of the Ninth International Conference on the Entity-Relationship Approach (ER'90)*, ed. H. Kangassala, 1. Amsterdam: North-Holland.

Borgida, A. 1995. "Description Logics in Data Management." *IEEE Transactions on Knowledge and Data Engineering* 7, no. 5: 671–682.

Borgida, A., R. Brachman, D. McGuiness, and L. Resnick. 1989. "CLASSIC: A Structural Data Model for Objects." In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, ed. J. Clifford, B. Lindsay, and D. Maier, 58–67. New York: ACM Press.

Brachman, R. 1979. "On the Epistemological Status of Semantic Networks." In *Associative Networks: Representation and Use of Knowledge by Computers*, ed. N. Findler, 3–50. New York: Academic Press.

Brachman, R., and H. Levesque, eds. 1984. *Readings in Knowledge Representation*. Los Altos, CA: Morgan Kaufmann.

Brodie, M. 1984. "On the Development of Data Models." In *On Conceptual Modeling: Perspectives from Artificial Intelligence*, ed. M. Brodie, J. Mylopoulos, and J. Schmidt, 19–47. New York: Springer.

Brodie, M., and J. Mylopoulos, eds. 1986. *On Knowledge Base Management Systems: Perspectives from Artificial Intelligence and Databases.* New York: Springer-Verlag.

Brodie, M., J. Mylopoulos, and J. Schmidt, eds. 1984. *On Conceptual Modeling: Perspectives from Artificial Intelligence, Databases and Programming Languages*. New York: Springer-Verlag.

Brodie, M., and S. Zilles, eds. 1981. *Proceedings of Workshop on Data Abstraction, Databases and Conceptual Modeling.* New York: ACM Press.

Bubenko, J. 1980. "Information Modeling in the Context of System Development." In *Proceedings of the IFIP Congress '80*, ed. S. Lavington, 395–411. Amsterdam: North-Holland.

Calvanese, D., M. Lenzerini, and D. Nardi. 1998. "Description Logic for Conceptual Modeling." In *Logics for Databases and Information Systems*, ed. J. Chomicki and G. Saake, 229–263. Dordrecht, Netherlands: Kluwer.

Chen, P. 1976. "The Entity-Relationship Model: Towards a Unified View of Data." *ACM Transactions on Database Systems* 1, no. 1: 9–36.

Chung, L., B. Nixon, and E. Yu. 1996. "Dealing with Change: An Approach Using Non-functional Requirements." *Requirements Engineering* 1, no. 4: 238–260.

Chung, L., B. Nixon, E. Yu, and J. Mylopoulos. 1999. *Non-functional Requirements in Software Engineering.* Norwell, MA: Kluwer.

Codd, E. 1970. "A Relational Model for Large Shared Data Banks." *Communications of the ACM* 13, no. 6: 377–387.

Codd, E. 1979. "Extending the Database Relational Model to Capture More Meaning." *ACM Transactions on Database Systems* 4, no. 4: 397–434.

Codd, E. 1982. "Relational Database: A Practical Foundation for Productivity." *Communications of the ACM* 25, no. 2: 109–117.

Collins, A., and E. Smith. 1988. *Readings in Cognitive Science: A Perspective from Psychology and Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann.

Conklin, J., and M. Begeman. 1988. "gIBIS: A Hypertext Tool for Exploratory Policy Discussion." *ACM Transactions on Office Information Systems* 6, no. 4: 281–318.

Copeland, G., and D. Maier. 1984. "Making Smalltalk a Database System." In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (*SIGMOD'84*), ed. B. Yormark, 316–325. New York: ACM Press.

Dahl, O.-J., and K. Nygaard. 1966. "SIMULA—An ALGOL-Based Simulation Language." *Communications of the ACM* 9, no. 9(September): 671–678.

Dahl, O.-J., B. Myrhaug, and K. Nygaard. 1970. "SIMULA 67 Common Base Language," Report S-22, Norwegian Computing Center, Oslo, Norway.

Dardenne, A., A. van Lamsweerde, and S. Fickas. 1993. "Goal-Directed Requirements Acquisition." *Science of Computer Programming* 20: 3–50.

Darimont, R., and A. van Lamsweerde. 1996. "Formal Refinement Patterns for Goal-Driven Requirements Elaboration." In *Proceedings of the Fourth ACM SIGSOFT Foundations of Software Engineering (FSE96)*, ed. D. Garlan, 179–190. New York: ACM Press.

DeGiacomo, G., Y. Lespérance, and H. Levesque. 1997. "Reasoning about Concurrent Execution, Prioritized Interrupts, and Exogenous Actions in the Situation Calculus." In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, ed. M. Pollack, 1221–1226. San Francisco: Morgan Kaufmann.

De Marco, T. 1979. *Structured Analysis and System Specification*. Upper Saddle River, NJ: Prentice Hall.

Dubois, E., J. Hagelstein, E. Lahou, F. Ponsaert, and A. Rifaut. 1986. "A Knowledge Representation Language for Requirements Engineering." *Proceedings of the IEEE* 74, no. 10: 1431–1444.

Feather, M. S. 1987. "Language Support for the Specification and Development of Composite Systems." *ACM Transactions on Programming Languages and Systems* 9, no. 2(April): 198–234.

Findler, N., ed. 1979. *Associative Networks: Representation and Use of Knowledge by Computers.* New York: Academic Press.

Fowler, M., and K. Scott. 1997. *UML Distilled.* Reading, MA: Addison-Wesley.

Galbraith, J. 1973. *Designing Complex Organizations*. Reading, MA: Addison-Wesley.

Gotel, O., and A. Finkelstein. 1995. "Contribution Structures." In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, 100–107. Los Alamitos, CA: IEEE Computer Society.

Greenspan, S. 1984. "Requirements Modeling: A Knowledge Representation Approach to Requirements Definition." Ph.D. diss., Department of Computer Science, University of Toronto.

Greenspan, S., A. Borgida, and J. Mylopoulos. 1986. "A Requirements Modeling Language and Its Logic." *Information Systems* 11, no. 1: 9–23.

Greenspan, S., J. Mylopoulos, and A. Borgida. 1982. "Capturing More World Knowledge in the Requirements Specification." In *Proceedings of the Sixth International Conference on Software Engineering (ICSE)*, 225–234. Los Alamitos, CA: IEEE Computer Society.

Hammer, M., and D. McLeod. 1981. "Database Description with SDM: A Semantic Data Model." *ACM Transactions on Database Systems* 6, no. 3: 351–386.

Hull, R., and R. King. 1987. "Semantic Database Modeling: Survey, Applications and Research Issues." *ACM Computing Surveys* 19, no. 3: 201–260.

Jackson, M. 1978. "Information Systems: Modeling, Sequencing and Transformation." In *Proceedings of the Third International Conference on Software Engineering (ICSE)*, 72–81. Los Alamitos, CA: IEEE Computer Society.

Jackson, M. 1983. *System Development*. Upper Saddle River, NJ: Prentice Hall.

Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard. 1992. *Object-Oriented Software Engineering —A Use Case Approach.* Reading, MA: Addison-Wesley.

Klas, W., and A. Sheth, eds. 1994. "Metadata for Digital Data." Special issue, *ACM SIGMOD Record* 23, no. 4.

Kramer, B., and J. Mylopoulos. 1991. "A Survey of Knowledge Representation." In *The Encyclopedia of Artificial Intelligence*, 2nd ed., ed. S. Shapiro, 743–759. New York: Wiley.

Lee, J. 1991. "Extending the Potts and Burns Model for Recording Design Rationale." In *Proceedings of the Thirteenth International Conference on Software Engineering*, 114–125. New York: ACM Press.

Lespérance, Y., T. Kelley, J. Mylopoulos, and E. Yu. 1999. "Modeling Dynamic Domains with ConGolog." In *Proceedings of the Eleventh Conference on Advanced Information Systems Engineering (CAiSE'99)* (Lecture Notes in Computer Science 1626), ed. M. Jarke and A. Oberweis, 365–380. Heidelberg, Germany: Springer.

Levesque, H. 1986. "Knowledge Representation and Reasoning." In *Annual Review of Computer Science*, Vol. 1, ed. J. Traub, B. Grosz, B. Lampson, and N. Nilsson, 255–287. Palo Alto, CA: Annual Reviews.

Loucopoulos, P., and R. Zicari, eds. 1992. *Conceptual Modeling, Databases and CASE: An Integrated View of Information System Development*. New York: Wiley.

MacLean, A., R. Young, V. Bellotti, and T. Moran. 1991. "Questions, Options, Criteria: Elements of Design Space Analysis." *Human-Computer Interaction* 6, nos. 3–4: 201–250.

Madhavji, N., and M. Penedo, eds. 1993. "Evolution of Software Processes." Special section, *IEEE Transactions on Software Engineering* 19, no. 12: 1125–1170.

Maida, A., and S. Shapiro. 1982. "Intensional Concepts in Propositional Semantic Networks." *Cognitive Science* 6: 291–330.

Minsky, M. 1975. "A Framework for Representing Knowledge." In *The Psychology of Computer Vision*, ed. P. Winston, 211–277. Cambridge, MA: MIT Press.

Mintzberg, H. 1979. *The Structuring of Organizations*. Upper Saddle River, NJ: Prentice Hall.

Mylopoulos, J., P. Bernstein, and H. Wong. 1980. "A Language Facility for Designing Data-Intensive Applications." *ACM Transactions on Database Systems* 5, no. 2: 185–207.

Mylopoulos, J., and M. Brodie, eds. 1988. *Readings in Artificial Intelligence and Databases*. San Francisco: Morgan Kaufmann.

Peckham, J., and F. Maryanski. 1988. "Semantic Data Models." *ACM Computing Surveys* 20, no. 3: 153–189.

Pfeffer, J., and G. Salancik. 1978. *The External Control of Organizations: A Resource Dependency Perspective*. Harper and Row.

Potts, C. 1997. "Requirements Models in Context." In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, 103. Los Alamitos, CA: IEEE Computer Society.

Potts, C., and G. Bruns. 1988. "Recording the Reasons for Design Decisions." In *Proceedings of the Tenth International Conference on Software Engineering*, 418–427. Los Alamitos, CA: IEEE Computer Society.

Quillian, R. 1968. "Semantic Memory." In *Semantic Information Processing*, ed. M. Minsky, 227–270. Cambridge, MA: MIT Press.

Rolland, C., and C. Cauvet. 1992. "Trends and Perspectives in Conceptual Modeling." In *Conceptual Modeling, Databases and CASE: An Integrated View of Information System Development*, ed. P. Loucopoulos and R. Zicari, 27–48. New York: Wiley.

Roman, G.-C. 1985. "A Taxonomy of Current Issues in Requirements Engineering." *IEEE Computer* 18, no. 4: 14–23.

Ross, D. 1977. "Structured Analysis (SA): A Language for Communicating Ideas," in "Requirements Analysis," special issue, *IEEE Transactions on Software Engineering* 3, no. 1: 16–34.

Ross, D., and A. Schoman. 1977. "Structured Analysis for Requirements Definition," in "Requirements Analysis," special issue, *IEEE Transactions on Software Engineering* 3, no. 1: 6–15.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design.* Upper Saddle River, NJ: Prentice Hall.

Schmidt, J., and C. Thanos, eds. 1989. *Foundations of Knowledge Base Management*. New York: Springer Verlag.

Scott, W. 1987. *Organizations: Rational, Natural or Open Systems.* 2nd ed. Upper Saddle River, NJ: Prentice Hall.

Shlaer, S., and S. Mellor. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data.* Upper Saddle River, NJ: Prentice Hall.

Solvberg, A. 1979. "A Contribution to the Definition of Concepts for Expressing Users' Information System Requirements." In *Proceedings of the International Conference on the E-R Approach to Systems Analysis and Design*, ed. P. Chen, 381–402. Amsterdam: North-Holland.

Thayer, R., and M. Dorfman. 1990. *System and Software Requirements Engineering.* 2 vols. Los Alamitos, CA: IEEE Computer Society.

Tsichritzis, D., and F. Lochovsky. 1982. *Data Models.* Upper Saddle River, NJ: Prentice Hall.

van Lamsweerde, A., R. Darimont, and P. Massonet. 1995. "Goal Directed Elaboration of Requirements for a Meeting Scheduler: Problems and Lessons Learnt." In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, 194–203. New York: IEEE Computer Society.

Vernadat, F. 1996. *Enterprise Modeling and Integration.* London: Chapman and Hall.

Webster, D. 1987. "Mapping the Design Representation Terrain: A Survey." Technical Report STP-093-87, Microelectronics and Computer Corporation, Austin, TX.

Widom, J. 1995. "Research Problems in Data Warehousing." In *Proceedings of the Fourth Conference on Information and Knowledge Management*, ed. N. Pissinou, A. Silberschatz, E. Park, and K. Makkai, 25–30. New York: ACM Press.

Yu, E. 1993. "Modeling Organizations for Information Systems Requirements Engineering." In *Proceedings of the IEEE International Symposium on Requirements Engineering*, 34–41. Los Alamitos, CA: IEEE Computer Society Press.

Yu, E., and J. Mylopoulos. 1994. "Understanding 'Why' in Software Process Modeling, Analysis and Design." In *Proceedings of the Sixteenth International Conference on Software Engineering*, 159–168. Los Alamitos, CA: IEEE Computer Society.

Yu, E., J. Mylopoulos, and Y. Lespérance. 1996. "AI Models for Business Process Re-engineering." *IEEE Expert* 11, no. 4: 16–23.

Zdonik, S., and D. Maier, eds. 1989. *Readings in Object-Oriented Databases.* San Francisco: Morgan Kaufmann.

2 Metamodeling

Matthias Jarke, Ralf Klamma, and Kalle Lyytinen

Metamodels—models of models—are intended to span the diversity of information systems environments. In this chapter, we review metamodeling and its applications using two frameworks. First, the International Organization for Standardization's Information Resource Dictionary System suggests a basic way of thinking about the generation, integration, and transformation tasks involved in developing and evolving coherent distributed information systems. Second, a "diamond framework" emphasizes four possible foci of metamodeling—ontology, notation, process, and goal—with respect to their support for system interoperation and adaptability. Five examples of metamodeling standards and tools illustrate the wide variety of challenges and approaches available today.

2.1 Introduction

Since the Middle Ages, and revived through mathematicians such as Russell and Gödel in the first part of the twentieth century, metaphysics, metaknowledge, and metalogic have captured the attention of philosophers and mathematicians. "Meta-x" can be read as "x about x" or "x behind x." Thus, metalevel techniques support abstract principles behind certain phenomena, from a specific viewpoint.

Not surprisingly, as the diversity and size of databases and information systems grow, we find an increasing need for *metadata*, that is, data about data. Although such metadata have traditionally been represented informally (e.g., in handbooks or data dictionaries), their automated analysis and manipulation requires their formal representation in models. Several major software vendors are developing or offering metadatabase products, also called repositories (Bernstein and Dayal 1994).

Repository data can differ across distributed information systems and can change over time. Thus, interoperability and adaptability require models about models: *metamodels*. Many application areas, including schema and data integration (Bernstein 2001; Catarci and Lenzerini 1993; Calvanese et al. 2001), reusable-component libraries (Constantopoulos et al. 1995), multimedia object modeling (Klas and Schrefl

1995), mechanical engineering (Pratt 2001), chemical engineering (Marquardt 1996), business process engineering (Scheer 1998; Nissen et al. 1996; Koubarakis and Plexousakis 2002), and even method engineering at the meta-metalevel (Iivari and Kerola 1983; Kumar and Welke 1992; Hong, Brinkkemper, and Harmsen 1995; Kelly, Lyytinen, and Rossi 1996) profit from the ability of metamodels to describe a modeling domain in a highly abstract manner.

In terms of data model theory, metamodels are related to the *instantiation* dimension of semantic data model abstractions (Brodie, Mylopoulos, and Schmidt 1984; Motschnig-Pitrik and Mylopoulos 1992). As classes abstract from instances, metaclasses abstract from classes, meta-metaclasses (M2 classes) abstract from metaclasses, and so forth.

Metamodels offer elementary *functionalities* such as (1) defining collections of semantically related classes, (2) grouping attributes according to different roles or facets, (3) defining shared methods or constraints over all instances of the classes contained in a metaclass, (4) defining shared class-level methods, attributes, and constraints, and (5) defining a common terminology framework to bridge semantic and terminology deviations among discrete domain-related information systems. Metamodeling environments use such functionalities to provide six major kinds of *services* to users and developers of information systems (Baumeister 1996):

- *Introspection* and *reflection* provide information about models, because metamodels are also models (Weyhrauch 1980). This service is mainly used in metaprogramming. In expert systems, it serves as a basis for explanation and control, for example, in Teiresias (Davis and Lenat 1982). In relational databases, system relations give information about the available data structures. A few object-oriented programming languages also support introspection and adaptivity in metamodels. For example, the Common Lisp Object System (CLOS) Meta Object Protocol (Kiczales, des Rivieres, and Bobrow 1991) offers a programmable metaclass level through which one can control and monitor the behavior pattern of the language without modifying existing application code. Similar facilities are also offered by logic programming languages such as Gödel (Hill and Lloyd 1989). Attempts to extend two-level object languages such as C++ with metadata also aim at these goals: Johnson and Palaniappan (1993) make a specifiable amount of type information about classes accessible at run time, whereas Chiba and Masuda (1993) present a simple metaobject protocol facilitating the change of method-calling conventions.
- *Adaptable modeling languages* allow method engineers to change the definition and behavior of models. Information systems development methods and tools have to be adapted to development contingencies such as uncertainty, size, time, resources, and available skills (Davis 1982; Necco, Gordon, and Tsai 1987; Olle et al. 1991). Moreover, all of these are likely to evolve over time across different projects. Several sur-

veys (Russo, Wynekoop, and Waltz 1995; Fitzgerald 1995) call for higher levels of adaptability. A case reported in Nissen et al. 1996 shows a need for more adaptable tools when an informal method is extended with technical support, but at the same time, the method must remain forward-adaptable to users' future needs. Nokia found it had to modify modeling techniques and processes constantly while a standardized system development method was being introduced in order to accommodate demands posed by large telecommunications applications (Aalto 1993). The developers of UML (Rumbaugh, Jacobson, and Booch 1999) have acknowledged the need for method adaptation by introducing the notion of stereotypes.

• *Focused model interaction* can be supported using metamodels as overlapping abstractions of heterogeneous modeling formalisms. This was studied initially in the context of heterogeneous database interoperability and information systems evolution. For example, the VODAK system (Klas and Schrefl 1995) uses a metamodeling extension of a fully object-oriented database for the integration of, for example, hypertext structures with traditional structured databases and multimedia information as well as for defining certain language extensions. More ad hoc integration formalisms, driven by applications such as distributed manufacturing, include Wiederhold and Genesereth's (1995) mediator approach, instantiated, for example, in systems such as GNOSIS (Gaines et al. 1996). Recently, formal metamodels based on description logics have been proposed as a basis for the integration of schemas in fields such as data warehouses (Calvanese et al. 2001) and bioinformatics (Goble et al. 2001). At a more abstract level, the resolution of modeling conflicts across multiple heterogeneous viewpoints has been the subject of much interest in distributed cooperative software engineering; see, for instance, the special issue of *IEEE Transactions on Software Engineering* devoted to the subject (Ghezzi and Nuseibeh 1998/1999).

• *Resource identification and indexing* can be supported by metamodels that assist search engines in locating relevent network-accessible information on the World Wide Web (WWW) or in distributed digital libraries. Recent advances in these areas propose various metamodeling facilities whose purpose is to collate, summarize, and filter metadata, primarily about network-available documents. For example, the Internet Anonymous FTP Archive (IAFA) template is a format for indexing information that can be used to describe various Internet resources. It encodes pieces of metadata as a record of attribute-value pairs such as "title:" "author:" "topic:" "abstract:" etc. (Deutch and Emtage 1994). Harvest's Summary Object Interchange Format (SOIF) contains a content summary for each information object that the Harvest gatherer collects from information resources in the network (Bowman et al. 1995). SOIF provides a means of abstracting collections of summary objects, allowing Harvest brokers to retrieve SOIF content summaries for many information

objects in a single compressed stream. The Dublin Core (Lagoze 1996) is a set of metadata elements introduced to describe, for discovery purposes, the essential features of networked documents in distributed digital libraries. In the design of the Core metamodel, consideration was given to mappings between the metamodel's elements and those of more specialized systems, such as library cataloguing. The MPEG-7 standard (named for the Moving Pictures Expert Group, which developed it) is a much more sophisticated metamodel aiming at the description of metadata for multimedia content (Avaro and Salembier 2001).

- *Context adaptation* of information delivery can be supported by context metamodels that allow the description of user interests, user location, user tasks, and other aspects needed for the personalization of information delivery (Riecken 2000). On a broader scale, context metamodels also include awareness (Gross and Specht 2001; Prinz 1999) and traceability (Ramesh and Jarke 2001) metamodels, which keep users aware of their present social context and collaboration history. An important aspect of metamodeling is to provide a means of tracing and sharing reasons for changes in ontologies, inspections, and reflections (Rossi et al. 2004) and to offer reusable components and patterns for intervening and modeling specific development situations (Zhang and Lyytinen 2001).
- *Distributed organizational cognition* can be supported through the building and adaptation of metamodels. Structured metamodels offer a means of sharing, inquiring, and transforming multiple ways of making sense of and organizing organizational experience (Tolvanen 1998). In this sense metamodels (and their organizing features) form important *boundary objects* (Carlile 2002) through which different communities of practice related to information systems (IS) design can take the perspectives of the others and at the same time make their perspectives known to the other communities (Boland and Tenkasi 1995).

In this chapter, we first employ two frameworks for understanding the principles and uses of metamodeling. To provide a basic *structural understanding* of metamodeling environments, section 2.2 describes an elementary architecture provided by the ISO Information Resource Dictionary System (ISO/IEC 1990) and sketches its various interpretations, applications, and formalizations. Then, starting from the observation that any metamodel is just one of many possible abstractions of a model, section 2.3 organizes the *universe of metamodeling techniques* according to a "diamond framework" of notation, ontology, process, and goal (Jarke et al. 1998). For each of these elements of the framework, we briefly survey metamodel uses for adaptation and interoperation. In section 2.4, we briefly describe a number of specific metamodeling standards and systems from research and practice, and section 2.5 summarizes with some general observations and challenges.

2.2 The IRDS Metamodeling Framework

Traditional modeling languages impose a certain worldview—a predefined linguistic framework—designed with a specific ontology of domain abstractions in mind. Examples of worldviews include entities and relationships in entity-relationship diagrams, processes and data flows in structured analysis, and objects and messages in object-oriented approaches. Although such a predefined worldview makes it easier to model a particular aspect of an application domain, it limits the application of the particular language to exactly this aspect. Therefore, most modern modeling methods—for example, object-oriented modeling techniques such as UML (Rumbaugh, Jacobson, and Booch 1999)—use multiple modeling notations together to address the different facets of a problem domain.

Managing large models with different notations poses serious problems of inconsistency, incompleteness, evolution, and reuse. Conceptual modeling languages incorporate ideas from knowledge representation, databases, and programming languages to provide the formal foundations necessary for system development at a level that is not too complex for people to understand and use yet is precise and simple enough for automated reasoning about the relationships between models (Brodie, Mylopoulos, and Schmidt 1984).

Metamodeling goes one step further: It allows one to customize *modeling formalisms* to the habits of individual modelers and users and to *specify the relationships* among models expressed in different modeling formalisms or domain ontologies. Even two decades ago, Kotteman and Konsynski (1984) had shown that four levels of instantiation are necessary to integrate the modeling of the usage and the evolution of information systems. A similar observation underlies the architecture of the International Organization for Standardization (ISO) Information Resources Dictionary System (IRDS) Standard (ISO/IEC 1990) depicted in figure 2.1 and explained in the following.

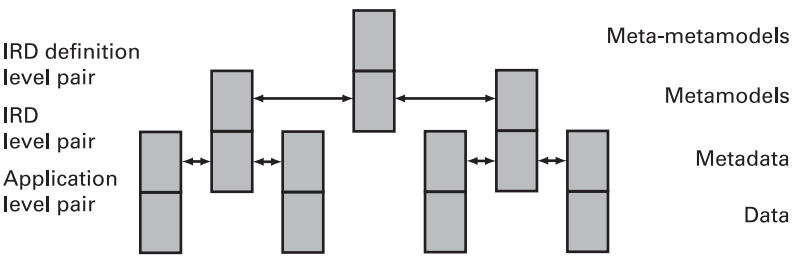


Figure 2.1
Interlocking level pairs in the ISO IRDS Standard

2.2.1 Levels and Interlocking Level Pairs

The IRDS architecture is intended to *interlock distributed application usage with distributed application development*. To achieve this purpose, information is organized in four levels of instantiation that we describe from bottom to top:

- The *application level* includes application *data* and program execution traces. This corresponds to the instance level of class-based languages.
- The *Information Resources Dictionary (IRD) level* includes *metadata*, that is, database schemata and application programs, plus any intermediate specifications, and also specifications of noncomputerized activities (e.g., workflows). It can also contain traces of development processes interlinking these specifications. This corresponds to the class level of class-based languages.
- The *IRD definition level* specifies *metamodels*: the languages in which schemata, application programs, and specifications are expressed. It may also contain the specification of possible static and dynamic interrelationships among these languages, for instance, design process models. This corresponds to the metaclass level of class-based languages.
- Finally, the *IRD definition schema level* specifies a *meta-metamodel* according to which the IRD definition level objects can be described and interlinked. Meta-metamodels thus define the language for method engineering.

As the figure shows, these four levels are grouped into interlocking level pairs. A level pair can be intuitively understood as a database in which the upper level is the schema and the lower level the database state. The architecture interlocks level pairs, in that the schemas of level pairs at one level can be coordinated by the database state of a level pair (dictionary) at the next higher level, thus creating a distributed database:

- *Application level pairs* correspond to traditional *application databases*, consisting of a schema and a database state.
- *IRD level pairs* correspond to data dictionaries, metadatabases, or *repositories*. At run time, they can serve as coordinators for distributed systems (e.g., in the Open Distributed Processing trader approach). At system evolution time, they serve as design databases.
- *IRD definition level pairs* serve the same purpose for distributed *method engineering environments*, linking multiple heterogeneous data dictionaries or design environments.

Interlocked application level pairs and IRD level pairs form a *distributed application environment*, whereas interlocked IRD level pairs and IRD definition level pairs

form a *distributed development environment*. Thus, the architecture provides the principal concepts for integrating the usage and the evolution of distributed systems.

2.2.2 Exploiting the Level Pair Architecture

When applying the IRDS framework to interoperable information systems, it is of decisive importance how the interlocking of level pairs is operationally exploited. Specifically, we can interpret the framework shown in figure 2.1 downward, upward, and sideways:

- Reading down, the architecture supports the *generation* of heterogeneous distributed environments. The relationship between level $n + 1$ and level n of the architecture in figure 2.1 is then understood as a type-instance relationship; that is, level $n + 1$ defines a type system for descriptions at level n. Good examples are the code generation facilities offered within the Rational Rose UML modeling environment (www.rational.com/products/rose/), the MetaEdit+ method engineering environment developed at the University of Jyväskylä (Kelly, Lyytinen, and Rossi 1996), and the KSIMapper developed at the University of Calgary (Kremer 1996).
- Reading up, level $n + 1$ defines one *view* (of several possible) on a given distributed system at level n. This is consistent with the *prototype approach* to object-oriented modeling, which is exemplified in systems such as n-dim (Westerberg 1996). Prototype models insist that a metamodel has mainly the task of drawing attention to a particular aspect of a distributed reality and is something that may equally well be created *after* the individual models under it have been constructed as before. Similar notions of a posteriori lower-bound and upper-bound schemas in semistructured data models are being pursued in the Extended Markup Language (XML) research community (Abiteboul, Buneman, and Suciu 2000). Thus, the architecture supports the *interoperation* of heterogeneous information systems with associated cross-notational consistency checking. In the execution environment, this corresponds to the integration of data from multiple heterogeneous sources, as, for example, in data warehousing. In the design environment, it corresponds to the consolidation of multiple heterogeneous method viewpoints, as discussed in detail in Ghezzi and Nuseibeh 1998/1999 and Nissen and Jarke 1999.
- Reading sideways, the architecture supports the semiautomatic *transformation* or *mediation* between multiple level n representations using mappings defined at level $n + 1$. Vertical mappings include the forward (Lefering 1993) and reverse (Jeusfeld and Johnen 1995) engineering of entity-relationship specifications to and from relational-database schemata. Horizontal mappings at the application level include, for example, information brokering from heterogeneous source representations to personalized client representations (Jeusfeld and Papazoglou 1999). An example at

the design level is the switching between graph and matrix visualizations of relationships among design objects (Kelly, Lyytinen, and Rossi 1996), whereby models obtain *representation independence*. Other examples of horizontal mappings include data warehouse architectures (Jarke et al. 1999) and the context interchange project at MIT (Goh et al. 1999), as well as the vision of the Semantic Web (Berners-Lee, Hendler, and Lassila 2001).

2.2.3 Formalizing the Relationships between Levels

Regardless of whether the IRDS framework is used downward, upward, or sideways, the *formalism used at the IRD definition level* determines how much freedom and precision can be offered in syntax, semantics, and presentation when supporting a given distributed application, design, or method engineering environment. In the ISO IRDS Standard, the formalism is ISO SQL, with a specific M2-level SQL schema. Commercial systems such as ARIS (Scheer 1994) and RDD-100 (Alford 1992), and also the American National Standards Institute (ANSI) version of IRDS, employ an extended entity-relationship model.

This is not a bad choice, as recent studies have shown that such extended entity-relationship models can be mapped nicely to description logics, thus making this approach both user-intuitive and computationally tractable. It is not surprising, therefore, that a version of this approach, called DAML-OIL (Horrocks 2002), is also being considered as the metamodeling language for the Semantic Web, despite the fact that the problem of how to map models across multiple levels of instantiation has not yet been solved for this family of description logics. As a consequence, these approaches, like their precursors in the knowledge-based systems community, such as the KIF and KQML language underlying the Ontolingua server (Farquhar, Fikes, and Rice 1997), tend to replace a true metamodeling approach à la IRDS by using very large generalization hierarchies (“ontologies”) at the metadata level, where the organizing role of the metamodel is played by the upper levels of these hierarchies, the “upper ontology” (Staab et al. 2001). As an extreme, we can consider CYC’s “commonsense” metadata structure of about a million terms organized around such an upper ontology with limited formal support; as an experimental metadata management system, CYC has been used for enterprise information system integration in the Carnot project (Huhns et al. 1993).

The exploration of different applications for interoperable information systems and their development environments has led to the recognition that *domain-specific metamodels* can provide substantially more support than generic solutions. For supporting such applications, the standard approach is to hard-code a specific M2 model in the support environment. Indeed, as a result of theoretical and implementation-level difficulties such as Russell’s paradox, most existing languages separate meta-level and class level explicitly, thus limiting the number of metalevels to a given

constant—usually one, sometimes two. For example, the Microsoft Repository (Bernstein et al. 1999) encodes the meta-metamodel defined in the UML standard in an object-oriented Common Object Model (COM)–based formalism, whereas the ARIS Toolkit (Scheer 1994) encodes the so-called ARIS House family of metamodels in a hard-coded entity-relationship model. MetaEdit+ employs a sophisticated graph-oriented meta-metamodel called GOPPR (graph-object-property-relationship-role) that proves particularly helpful for the engineering of graph-based design methods (Kelly, Lyytinen, and Rossi 1996).

In contrast, an extensible metamodeling environment allows the syntactic, semantic, and user-oriented specification of M2 models within a single environment. Amalgamation has been proposed as a means of switching back and forth between object and metalevel in logic programming (Kowalski 1979). For example, the ConceptBase system (Jarke et al. 1995) supports a deductive, object-oriented language called Telos (Mylopoulos et al. 1990) that allows for an arbitrary number of metalevels. The problem of infinite regress is solved by reflexive instantiation at a so-called omega level, combined with careful semantic restrictions concerning the use of negation to avoid set-theoretic paradoxes (Jeusfeld 1992). Nissen and Jarke 1999 shows that this formalization can be extended even to the case of distributed modular modeling and operations environments. However, at a single level of analysis, the underlying logic, Datalog with stratified negation, is less powerful than the description logic formalism underlying, for example, DAML-OIL. To what degree the advantages of the two techniques can be combined with one another, and possibly with more dynamically oriented formalisms such as model checking of finite-state automata specifications, is still an open research issue. Initial pragmatic solutions have been attempted in requirements engineering (Fuxman et al. 2001; Gans et al. 2001).

2.3 Dimensions of Metamodeling

Any given metamodel enables the detection of different kinds of transformations and conflicts and other higher-level insights and development processes. In other words, as pointed out by philosophers (cf. Winograd and Flores 1986 for a detailed exposition), the metamodel used constrains the action and interaction of modelers and users alike.

However, the development of metamodels is not completely haphazard. In this section, we first present a framework according to which metamodeling approaches can be classified, then describe its application to interoperability and adaptability. We analyze metamodel-based interoperability and adaptability by developing a framework called the *diamond model* (see figure 2.2, from Jarke et al. 1998). The diamond model consists of two parts: (1) the triangle of ontologies, notations, and processes, which describes a possible space of choices when one is adapting or interrelating

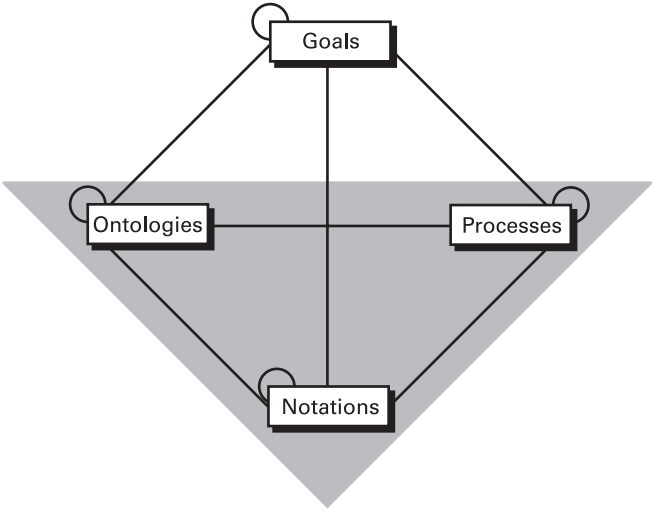


Figure 2.2
Four aspects of metamodel-based environments

models; and (2) the goals of an information systems development (ISD) project, which drive the choices made in the triangle.

The question each aspect in the triangle seeks to capture can be summarized as follows:

1. What are underlying ontologies of the domains handled in the information system?
2. What notations are used to capture and represent different aspects of the system and its environment?
3. What processes are enacted when models are derived, assessed, and validated?

Each of these aspects can be independently modeled and represented using a number of alternative representation schemes and mechanisms, which result in different types of metamodels. Consequently, metamodels can deal with representations of domain-related concepts and vocabularies and their organization into reference frameworks and metamodel hierarchies (*ontology-based metamodeling*), specifications of notations and notational systems (*notation-oriented metamodeling*), or specifications of large-grained processes or small-grained process chunks (*process-oriented metamodeling*).

The diamond shape of the model is intended to illustrate that the model's three aspects are neither exclusive nor orthogonal. Each viewpoint complements the other viewpoints, and all are required to yield "complete" interoperability and adaptability

of the modeled environment. For each of these aspects, we find that "islands of automation" have been built; that is, each aspect can be modeled to some extent independently of the others.

In the following subsections we examine each aspect of the model, its contribution to interoperability and adaptability, and its relationships to the other aspects (lines and circles in figure 2.2). Finally, we discuss how the different aspects interact in the evolution of information systems engineering methods.

2.3.1 The Ontological Aspect

It is impossible to represent a system in its full detail. Therefore, it is necessary to pay attention to a small number of abstractions based on a set of generic concepts. These concepts should be meaningful and sufficient to conceptualize the phenomena of the systems development domain and to imply some properties of how the system will operate in that domain. The enumeration (and sometimes formal specification) of the concepts used in the conceptualization is called an ontology (Gruber 1993).

An ontology consists of a set of concepts and their relationships, forming a conceptual structure that underlies the interpretation of any system model. Thus ontology is defined by basic terms and relationships comprising a vocabulary to represent a development area (Neches et al. 1991). All methods aimed at modeling systems are based on some ontology.

Two distinct views of ontology can be distinguished in the IS literature (Wand 1996). As a philosophical concept, an ontology forms something fundamental from which all universes of discourse can be formed. In the AI view of ontology, ontologies differ among domains.

The need to establish a fundamental ontology for IS is visible in suggestions of generic reference models for IS modeling (Wand and Weber 1989), which have recently been formalized as metamodels in Rosemann and Green 2002. As a result of differences in systems (from business administration systems to automated robots), development contingencies (users' familiarity with concepts), and different philosophical positions, there is no general agreement as to whether such a fundamental ontology does exist or whether it is really needed (see, e.g., Hirschheim, Klein, and Lyytinen 1995). Heated arguments concerning what ontological constructs entity-relationship modeling should include (entities, relationships, and/or attributes?) are examples of this lack of agreement.

Domain-based ontologies have focused on specific reference models such as

- the ARIS House family of metamodels for business integration (Scheer 1998);
- the STEP family of standards for product planning and design (Pratt 2001);
- similar standards in telecommunication, medicine, and elecronic commerce (Brodie 1997).

The domain theory developed for requirements engineering in the European NATURE project can be cited as one of the few attempts to study the interaction between fundamental and domain-based ontologies (NATURE Team 1996).

The ontological constructs that can be defined vary from IS environment to IS environment largely depending on the semantic power of the metamodeling language used (Harmsen and Saeki 1996). Here semantic power may vary from that in simple entity-relationship-based formalisms like those in System Encyclopedia Manager (ISDOS 1981) up through that offered by powerful object-oriented modeling notation like GOPRR (Kelly, Lyytinen, and Rossi 1996; Tolvanen 1998). Languages like Telos also include powerful inference mechanisms that offer a rich set of horizontal and vertical integrity constraints (Jarke et al. 1995; Nissen et al. 1996). Others offer specific procedural languages for specifying consistency rules and transformations (Boloix, Sorenson, and Tremblay 1991). Recent ontology language formalisms developed in the context of the Semantic Web initiative are described in Gomez-Peres and Corcho 2002.

Besides the choice of representation language, careful adherence to some basic distinctions and principles of ontology construction has also proven critical to success in ontology management. An overview of such principles is given by Guarini and Welty (2002). Examples include rules for distinguishing among seemingly similar abstraction mechanisms such as generalization, instantiation, disjunction, and part-of relationships.

Another dimension on which ontological constructs can be analyzed is the scope and power of their integration mechanisms. Most metamodeling languages cover only a singular technique and contextual mappings between notations and ontologies (Smolander 1992). Some environments offer richer mechanisms than others for specifying levels of integration between techniques (ter Hofstede and van der Weide 1993), whereas some can also handle *m*:*n* mappings between ontologies and notations (Kelly, Lyytinen, and Rossi 1996).

Generally, the more mature an application domain is, the greater the number of stable ontologies available for that domain. For example, in electrical engineering, the concepts for interpreting and analyzing electrical circuits are widely shared and standardized. Suggested standards for computer-aided software engineering (CASE) interoperability are also based on a fixed metamodel. These include Case Data Interchange Format (1994), used in transferring repository data among CASE tools, and STEP, which supports integration of production data (Pratt 2001). IS development, in contrast, because of its youth, generally lacks standardized fundamental ontologies, despite efforts to develop such standardized ontologies in general (Oei and Falkenberg 1994) and in systems modeling (e.g., UML [Booch, Rumbaugh, and Jacobson 1998]).

Development and integration of domain ontologies requires mechanisms for combining and managing sets of interrelated vocabularies. Research into developing such mechanisms has been carried out mainly by investigating principles and theories of ontology hierarchies consisting of a shared ontology (fundamental ontological constructs representing "deep" domain structures, such as data flows), customized ontologies (domain-dependent extensions to fundamental ontologies, e.g., time extensions to data flow diagrams [DFDs]), and application- or domain-related assertions (state-dependent knowledge of the system to be developed, e.g., integrity constraints defined over specific processes in DFDs in some domain) (Neches et al. 1991). A good example of the integration of ontologies is the domain of medicine, in which the Unified Medical Language System (UMLS) is employed as a top ontology for integrating more specialized medical-domain ontologies (National Library of Medicine 1994).

Most CASE tools offer a fixed set of modeling techniques and associated ontologies, because of the lack of a specific metamodeling layer in which the data semantics are differentiated from the data representations. Ontologies can be made adaptable through three alternative mechanisms, presented here in order of decreasing degrees of freedom to specify new ontologies:

- by providing means to specify new metamodels and expand the data semantics
- by providing means to extend an existing metamodel
- by providing means to select a subset of an existing large reference metamodel

Meta-CASE tools contain some declarative mechanisms for defining metamodels. Examples of such meta-CASE tools include MetaView (Sorenson, Tremblay, and McAllister 1988), ConceptBase (Jarke 1992), RAMATIC (Bergsten et al. 1989), and MetaEdit+ (Kelly, Lyytinen, and Rossi 1996). Metamodel extensions are supported in older CASE tools like Excelerator (Marttiin et al. 1993). A reference model approach was used in some early CASE tools like PSL/PSA (Teichroew and Hershey 1977) and also in IBM's AD/Cycle model (Mercurio et al. 1990). It has gained increasing importance in meta-CASE tools like Maestro II (Merbeth 1991) and in tools that maintain SAP reference models (Scheer 1998; Hernandez 1997).

Management of change at the metamodel level (metamodel evolution) has been discussed to some extent in the method engineering literature (Harmsen, Brinkkemper, and Oei 1994; Marttiin, Harmsen, and Rossi 1996; Rossi et al. 2003). Conceptual foundations for managing such change include metamodeling hierarchies (Oei and Falkenberg 1994) and incremental method adaptations (Tolvanen and Lyytinen 1993; Tolvanen 1998). Current tools do not, however, provide many elegant solutions for managing metamodel evolutions. Most deal with them as simple model updates into the metamodel (see, e.g., strategies for metamodel management in

MetaEdit+ [Kelly, Lyytinen, and Rossi 1996]) and through incremental adaptation and reuse of the existing repository of metamodel components (Kelly, Lyytinen, and Rossi 1996; Tolvanen 1998; Kelly 1998). These methods have been lately enhanced with reusasable metamodel components that include context models and ports for easier pluggability (Zhang and Lyytinen 2001) and through an analysis of ways in which metamodel components can be made reusable. There are no graceful mechanisms for addressing consistency between the type and instance level data, and removal of a metamodel construct also removes, in all environments, the instance data from the repository, as no metalevel versioning and associated semantics are available.

2.3.2 The Notational Aspect

Any ontological construct can be manipulated or communicated only by using some notation. Notations can be classified according to their level of formality into formal (logic, rules), semiformal (structured and object-oriented notations), and free-form or informal notations (e.g., rich pictures in Checkland 1981). A notation's level of formality also characterizes to what extent it can be manipulated and new information derived from the representations using computer-based tools. Independent of its level of formality, a notation can also be classified, according to its presentation style, as graphical, matrix, textual, tabular, or form-based.

To be useful in representing a system or its environment in some domain, a notation must be associated with rules of ontological mapping (i.e., semantics). For example, a notation with graphical nodes and links might imply an ontology that distinguishes between objects and relationships but could also imply an ontology that distinguishes between states and activities. Usually higher degrees of formality imply stricter rules regarding how notational constructs are mapped into ontological constructs (and therefore fewer degrees of freedom in representing varied situations in the domain of representation). This is well formalized in formal logics through their model theory and associated axioms of model validity and satisfaction (Kleene 1967).

There is an *m*:*n* relation between notations and ontologies. Each construct in an ontology must be related to at least one notational element (a phrase, an icon, etc.) to enable its use, and it can be related to more than one notational element. There are also specific rules and theories that specify what constitute good mappings among ontological theories, ontological constructs, and their representations (Wand and Weber 1993, 1995). For example, the concept "Entity" can be related to a textual string "entity" or to a graphical rectangle. The assignment of more than one notational element to an ontological construct enables the same ontological construct to be presented in different situations and for stakeholders with different needs. The same construct can be mapped to different notational systems with different levels of

formality or with different presentation styles. In addition, a notation may be related to more than one ontological "system." Therefore, the interpretation of the notation (i.e., its mapping) requires knowledge of the context in which the notation is used. For example, a rectangle can represent an entity in data modeling, whereas it represents a terminator in the data flow view.

A variety of notations of varying complexity and formalization need to be available because systems development situations cover varying applications (e.g., real-time systems and business reengineering) with different notational traditions, project contingencies (e.g., developer skills and familiarity with notations vary), and different task demands (e.g., a move from informal to formal, from fuzzy to precise). For the same reasons, metamodeling capability is needed to integrate various notations and their semantics (through shared ontological elements or mapping between ontological elements).

In integrating notations, we distinguish between situations involving horizontal and vertical integration (Lehman and Turski 1987; Zelkowitz 1993). Vertical integration maps notational elements onto development stages and therefore requires tools and mechanisms for forward and reverse engineering and requirements tracing (Chen and Norman 1992). Horizontal integration enforces integrity in "horizontal" design representations by using integrity rules, transformations, and hypertext connectivity to link various notational elements.

Both types of integration situations have to be represented in metamodels by mapping notations to one another (cycle from notations to notations) and by mapping notations to different ontologies (relationships between notations and ontologies). Interpretations of notations within use contexts, that is, mappings between ontological constructs and notations, must be agreed upon within development tasks in which the ontology and notations are used (situational method engineering). In the case of a mature ontology, like that in electrical engineering, in which relations between the ontological concepts and notations are stable and presumed, the choice of the use context is unproblematic. In information system development, such fixed mappings between concepts and notation hardly exist despite standardization efforts, and they can be difficult to obtain because of the wide variation in modeling domains, types of systems developed, and intensive learning effects associated with system development (Tolvanen 1998).

Recent method engineering activities, such as the definition of the Unified Modeling Language (Booch, Rumbaugh, and Jacobson 1998), have aimed at establishing unified standards for notations but this unification goal may be unreachable because of the variety of development situations any such standards must cover. What we need instead of unified notation standards is a better (meta)modeling mechanism that allows flexible and multiple mappings between notations and ontologies and provides powerful tools for enforcing horizontal and vertical integration. Critical

here is also the ease and flexibility of achieving such mappings, which must be proportional to the benefits obtained from new mappings. This calls for flexibility and ease of use with meta-CASE environments.

As noted at the beginning of the section, ontological constructs can be manipulated only by using some kind of notation. Thus, all CASE tools must provide at least one notational construct for each ontological construct they distinguish. In early CASE tools such as PSL/PSA (Teichroew and Hershey 1977), the dominant presentation style was textual. Since the late 1980s, graphical notations have dominated. Traditional CASE tools use mostly fixed notations, whereas meta-CASE tools support varying levels of notational adaptability. The simplest form of notational adaptability involves offering a selection of a symbol from a set of graphical symbols or allowing a set of text aliases for a particular ontological construct. Such notational adaptability has been supported since the late 1980s in CASE tools such as the Excelerator tool family (Marttiin et al. 1993). Another form of notational adaptability involves allowing a change in any notation but those early approaches did not support changes in the presentation style, in particular changes to the graphical symbols. This requires that a dedicated metalevel graphical editor be available to define symbols. Rational Rose offers stereotypes for this purpose. Symbols can be defined in most meta-CASE tools, and some also offer several presentation styles. A separate editor is needed, however, for each presentation style (Kelly, Lyytinen, and Rossi 1996; Kelly 1998; Bergsten et al. 1989).

Difficulties in managing a variety of ontologies and notations simultaneously are recognized in the National Institute for Standards and Technology (NIST)/ European Computer Manufacturers Association (ECMA) model (Zelkowitz 1993). The associated manipulation of metalevel constructs also requires a notation or several notations. Metamodel manipulation has been performed, for example, textually in ConceptBase (Jarke 1992), graphically in MetaEdit (Smolander et al. 1991), and by using forms in MetaEdit+ (Kelly, Lyytinen, and Rossi 1996).

2.3.3 The Process Aspect

We define process as "a set of partially ordered process steps intended to produce a desired product" (Heineman et al. 1994). A process (meta)model is an abstraction of (a set of) processes. Process metamodels are closely related to both ontologies and notation in that any process model assumes a specifc ontology about processes and demands associated notations to represent, analyze, and enact them. Depending on goals, a process model's intended usage (e.g., by humans or tools, for process control or process guidance), its level of granularity and associated ontology, and the formalism used (notation) vary greatly (Finkelstein, Kramer, and Nuseibeh 1994; Curtis, Kellner, and Over 1992; Armenise et al. 1993; Lott 1993). Process models vary in their levels of formality; they can express a specific abstraction of process ontology

(organization, project, group, team, individual, etc.) and select a particular view (architectural, design, definitions, etc.) of that ontology. They can support (enforce, guide, control) process execution, assist in process analysis, aid in process understanding, or predict behavior.

What information is represented in process models depends thus on the underlying assumptions of the modelers (ontologies) and the goals they pursue (Lonchamp 1993). This is reflected in distinctions between descriptive and prescriptive metamodels (McChesney 1995). Descriptive models (e.g., traceability models) describe how an information system is (or has been) developed (Pohl 1996; Ramesh and Jarke 2001; Rossi et al. 2004). Prescriptive models define and control the means by which a task is to be completed and the order in which tasks are to be performed.

A number of relationships between processes and ontology-notation combinations have been studied. On the one hand, a process model (task) can determine specific concepts and their representations. On the other, the available ontologies and notations influence the choices that one can make in outlining a process model; for example, the Rational Unified Process has been strongly influenced by UML language constructs. To understand how metamodeling handles the relationships among processes, ontologies, and notations, we distinguish three kinds of process elements that behave differently in terms of their implications for modeling relationships between tasks and ontologies-notations:

1. Fundamental actions manipulate constructs that are used to model the system under development; that is, they adopt concepts from the underlying ontology as input and return them as output. Therefore, the process steps of the information system development method are primarily determined by the ontology chosen. If the notation for the method is not sufficient to express all the constructs in the ontology, one cannot model fundamental actions dealing with these constructs, either. For example, if the notation provides no elements to represent "relationship" in entity-relationship modeling, it is not possible to define a process step to create a relationship between two entities.

2. Notational actions deal with notational aspects of representations (cosmetics) without affecting the underlying conceptual structure. Examples of such activity abound: diagonalizing a matrix or designing a layout. Another example of notational actions is the transformation of a representation from one notation into another, when more than one notation is linked to the same ontology.

3. Process-related actions neither change concepts nor their representations but influence how processes are carried out. Examples are browsing through a decision tree or starting, stopping, or resuming a process fragment. In addition, activities orthogonal to uses of any method, like recording design rationales and document management (e.g., versioning), fall into this category.

Integrations of various process modeling languages (PMLs) are rare. The few exceptions discussed so far, such as Viewpoints (Sommerville et al. 1995), do not use metamodel-based techniques. The only metamodel-based exception we are aware of is a process metamodel developed in the NATURE project that—through the notion of context specialization—allows for the integration of different process model aspects such as decision support, workflow management, and hierarchical planning (Jarke et al. 1994; Pohl 1996; Rolland 1998; Koskinen 2000). The Process-Integrated Modeling Environment (PRIME) demonstrates the power of process integration for the guidance and traceability of engineering environments (Pohl et al. 1999).

An explicit process representation for process adaptability is now offered by many tools, including some configuration management tools (program files as products), workflow management tools (shared documents and reviews as products), and groupware tools. However, simultaneous adaptation of processes and products (ontology and notation) is not well managed in current tools.

In most CASE tools the process model is implicit and, if defined, fixed. Although some tools, like Maestro II (Merbeth 1991), provide an explicit process model, they offer only a set of predefined process models to choose from. Some approaches seek to substitute other constructs for a process model. These include the use of hypertext links (e.g., HyperCASE [Cybulski and Reed 1992]) or of agent models (Yu and Mylopoulos 1994). Adaptable process tools, on the other hand, assume fixed ontologies and notations or do not consider any specific deliverables (i.e., ontologies and notations are not defined). Examples of the first approach are EPOS (Jaccheri and Conradi 1993), and SPADE (Bandinelli, Fuggetta, and Ghezzi 1993). Examples of the second approach include CPCE (Lonchamp 1995), and SCALE (Oquendo 1995).

2.3.4 The Goal Aspect

Goals drive the way situational contingencies and higher-level goals (such as quality, economic value, and effectiveness) are mapped onto decisions about processes, their notations, ontologies, and their relationships (Castro, Kolp, and Mylopoulos 2002). For example, on the process side, a contingency of having two teams and at the same time the need for consistent and error-free designs requires the specification of a rigid process model with several coordination points. On the notational side, a contingency of using Smalltalk and the goal of having a parsimonious object class specification require that class diagrams refrain from using multiple inheritance.

Goal-based interoperation and adaptation are probably the least-studied aspects of metamodeling. The majority of IS methods do not distinguish between situational expectations and deviations in method use (Iivari and Kerola 1983; Vlasblom, Rijsenbrij, and Glastra 1995). Typically IS methods provide a fixed set of concepts and notations and some procedural guidelines without adaptation possibilities. Although

in some methods the need for adaptability has been recognized (cf. Booch 1994; Coleman et al. 1994), these methods neither include comprehensive descriptions of method use environments nor provide adequate mechanisms for modifying the method after the modification need has been recognized.

Some researchers have at least identified goals that drive method adaptation. We identify four approaches:

- *Goals of the ISD project* (costs, satisfaction, resources, and schedule): Goals related to situational methods are studied in the S3 (situation-scenario-success) model (Harmsen, Lubbers, and Wijers 1995) and those related to software process quality in the GQM (goal-question-metric) approach (Basili and Rombach 1988).
- *Changes in stakeholder's values*: Because use of methods and supporting tools is essentially the same kind of process as use of any information system, it is appropriate that methods meet users' requirements. Hence, methods should continuously satisfy the requirements of stakeholders—such as designers, programmers, IS users and managers (Kumar and Welke 1992). This is, in fact, an essential condition for the acceptance of methods, because method users will more easily learn methods, accept them, and actually use them when the method supports the continuous evolution of stakeholder's requirements.
- *Situational contingencies of the ISD project*: Contingency theory claims that there is no universally acceptable ISD method that is suitable for all circumstances. On the one hand, this justifies the adaptation of methods. On the other hand, contingency researchers (e.g., Davis 1982; Kotteman and Konsynski 1984; Sullivan 1985) have sought to identify characteristics (called *situation dependencies*) that control the outcomes of method use. These characteristics, such as the type of an information system, the programming language applied, development culture and process maturity, and developers' experience, can affect the adaptation of a particular method.
- *Accumulated knowledge of methods* (Jarke et al. 1994; Tolvanen 1998): Methods can hardly be adapted in one step because of the dynamics of method use: Organizations learn from experiences and become handy with methods as they are used. Accordingly, methods need to be adapted, managed, and maintained continually based on garnered experience, and there are several abstraction mechanisms available to improve the reuse of method-related knowledge (Zhang and Lyytinen 2001).

In recent metamodeling proposals, these approaches are addressed from three angles. First, explicit modeling of goals and means-ends relationships have been investigated since the early 1990s, for example, in the KAOS approach (Dardenne et al. 1993). Second, goal interoperation by multiple interdependent stakeholders is addressed in the i* formalism proposed by Yu (1995), which allows the description

of strategic interdependencies and strategic-goal hierarchies in a single framework. Finally, contingency issues and goals have been related to process metamodels via the construct of design rationale (Ramesh and Dhar 1992). Studies of requirements traceability show that rationale metamodels and mechanisms of varying sophistication are increasingly used in practice, and indeed sometimes mandated by procurement standards (Ramesh and Jarke 2001).

2.3.5 Method Engineering and Evolution: Interaction of the Aspects

In any given project, the bottom triangle in figure 2.2 suggests a set of development methods by defining the available array of ontologies, notations, and process steps. On top of this triangle, the current goal set defines a view that selects ontologies, notations, and processes and the relationships among them that are of current relevance. The goal set acts like a filter that drives choices among three method engineering aspects. Shifting a goal set in a project leads to the application of another view and necessitates the selection of process steps, ontological constructs, or representations from the pool of metamodels according to changed goals. We call a change of this type goal-driven method adaptation.

Some changes during a project cannot be accommodated solely by the adaptability offered in the current state of the metamodels. In other words, some changes call not only for making choices within the current diamond but also for an evolution of the diamond, in that new goals, processes, ontological constructs, notational elements, and their relationships are added into the metamodel base (Tolvanen and Lyytinen 1993). Method evolution can be carried out by the following alternative strategies that have varying degrees of impact on developers' work. To reduce disruptions in the development environment, the normal priorities among these strategies are as follows:

1. Change the processes by leaving the notation and the ontology stable.
2. Establish new relationships between existing notations and ontological concepts.
3. Introduce new notations for existing notations.
4. Introduce new ontologies (of varying scope).
5. Change the goal.

2.4 Examples of Metamodeling Environments

In this section, we illustrate how the various aspects of metamodeling and metadata management presented in the previous sections can be packaged into standards and tools for metamodeling. We have chosen five example systems that are grounded in research but also used in practice:

- The ARIS Toolkit is probably the current world market leader in business process representation for the enterprise resource planning (ERP) context, thus showing a prime example of an *ontology-focused* metamodel; a similar (and earlier) approach in a different domain along the same lines was the Unified Medical Language System meta-metamodel.
- The Microsoft Repository (recently renamed Metadata Engine) is shipped with most high-end versions of the Microsoft Office package and is closely linked to the current market-leading *notational* metamodel standard, UML.
- The MetaEdit+ environment is particularly strong in the *declarative* aspects of graph-based metamodeling and method engineering.
- The ConceptBase environment has its focus on the *access-oriented* and *integrative* aspects of metamodeling relying on a logic-based object-oriented formalism.
- The MPEG-7 multimedia metadata standard is an example of a *descriptive*, *access-oriented* metamodel on top of the XML standard.

2.4.1 Ontology-Centric Metamodeling: The ARIS House and Toolkit

Standardized ERP systems such as those marketed by SAP and Oracle were one of the most impressive successes in business application system standardization and integration in the 1990s. According to Scheer (1994), these systems' main advantage is the enormous wealth of knowledge encoded in them, which has radically reduced the business analysis efforts for standard administrative tasks in enterprises. In essence, ERP systems development is now largely a process of customizing standardized business ontologies rather than inventing new ones.

Although ERP products support these ontologies implicitly in their code, databases, and user interfaces, a number of modeling tools have been developed that make the underlying models explicit, thus offering a better understanding, design, and maintenance capability for complex ERP environments. Examples include the INCOME system offered by PROMATIS to assist in Oracle-based ERP systems applications (Oberweis et al. 1994) and the market-leading ARIS Toolkit developed by IDS Scheer (Scheer 1994).

The ARIS Toolkit offers a particularly intuitive metamodel structure that we briefly describe here. Considering that ERP systems are characterized more by enormous size and multiple perspectives of interest than by complexity in details of the algorithms or individual applications, Scheer (1998) proposed the ARIS House structure of metamodel collections, which is shown in figure 2.3.

Essentially, the ARIS House elegantly combines two dimensions of ERP systems engineering. One dimension concerns the phase structure of requirements definition (application oriented), the design specification (system oriented), and the technical

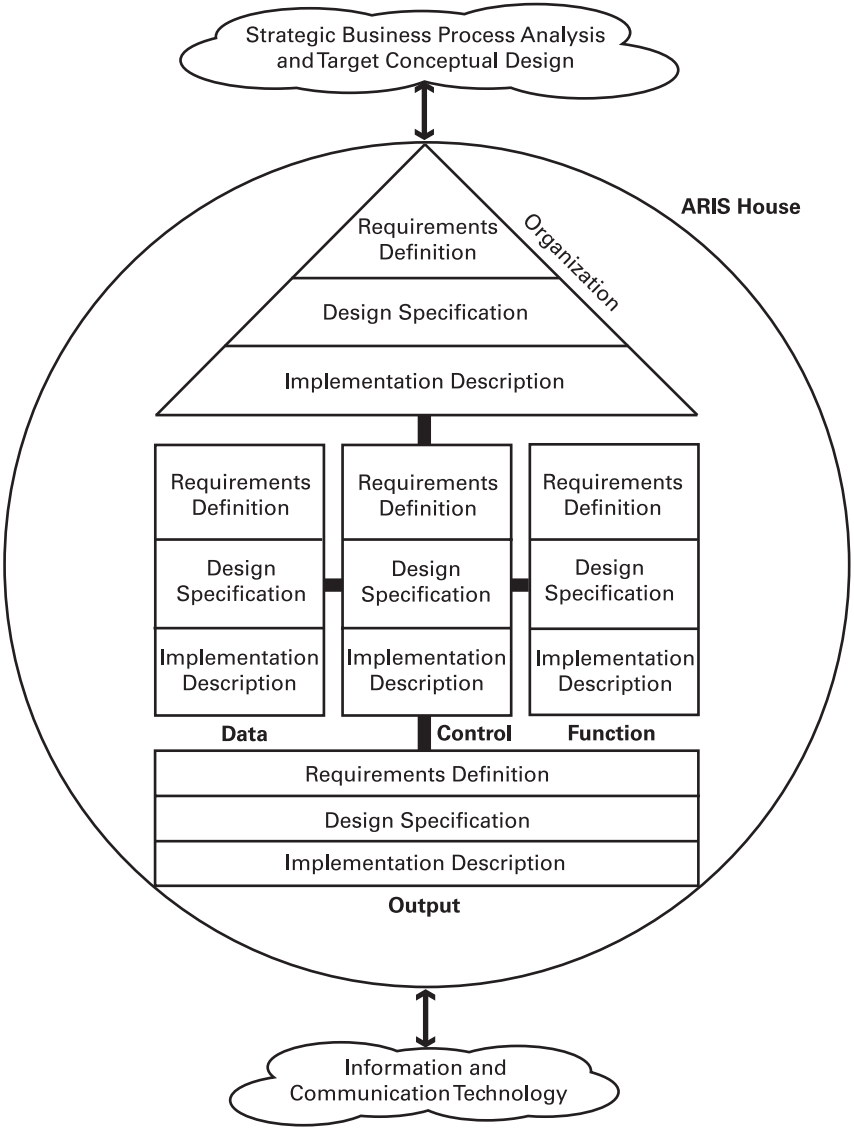


Figure 2.3
The ARIS House structure for collections of ERP metamodels. Adapted from (Scheer 1998)

implementation, as previously prototyped in the DAIDA metamodeling environment (Jarke 1993). The other dimension considers the well-known perspectives of integration in sociotechnical systems, namely, data, function, organization, output, and (for the integration itself) control.

The emphasis of ontology-based model development in the ARIS Toolkit is on reference models at the level of requirements engineering (application domain modeling). To describe in more detail the possible interrelationships among the five perspectives of data, function, output, organization, and control, Scheer has developed a semiformal meta-metamodel in an extended entity-relationship notation, as shown in figure 2.4.

The figure is organized around the notion of function. In the upper left of the figure, various inputs to the function concept are visible, and the upper right shows the corresponding outputs. On the same level as the function itself, we see the events that represent the control flow of the system. The context of the function is described by environment data and goals (in the upper center) as well as the related organizational unit. Other resource and organizational aspects are shown in the lower part of the figure.

The five aforementioned perspectives can be understood as overlapping views on this meta-metamodel (Scheer 1998). The metamodel overlaps could be used for purposes of transformation or consistency checking between models defined in the different perspectives (data, function, output, organization, control) if a formal, computationally tractable formalization of the meta-metamodel were available (e.g., one along the lines of the ConceptBase logic described in chapter 3). In practice, the ARIS Toolkit uses the meta-metamodel as an intuitive guideline for individual integrity checks or transformational functions without a seamless formal foundation. Nevertheless, the ARIS Toolkit is probably the most comprehensive ontology-based metamodeling environment for business applications in existence and is clearly the leader of the world market in this domain.

2.4.2 Notation-Centric Metamodeling: The Microsoft Repository

The Microsoft Repository (MSR) (Bernstein et al. 1999), marketed under the name of Metadata Engine, has a metamodeling language that is a combination of relational and object-oriented solutions. Although its underlying storage mechanism is relational, the data model is based on Microsoft's Common Object Model, a binary-object standard.

A main strategy of MSR has been the definition of a broad range of information metamodels for system domains relevant for Microsoft customers and original equipment manufacturers. To define the relationships among objects in a repository, MSR offers structural information and some object-oriented methods and abstraction

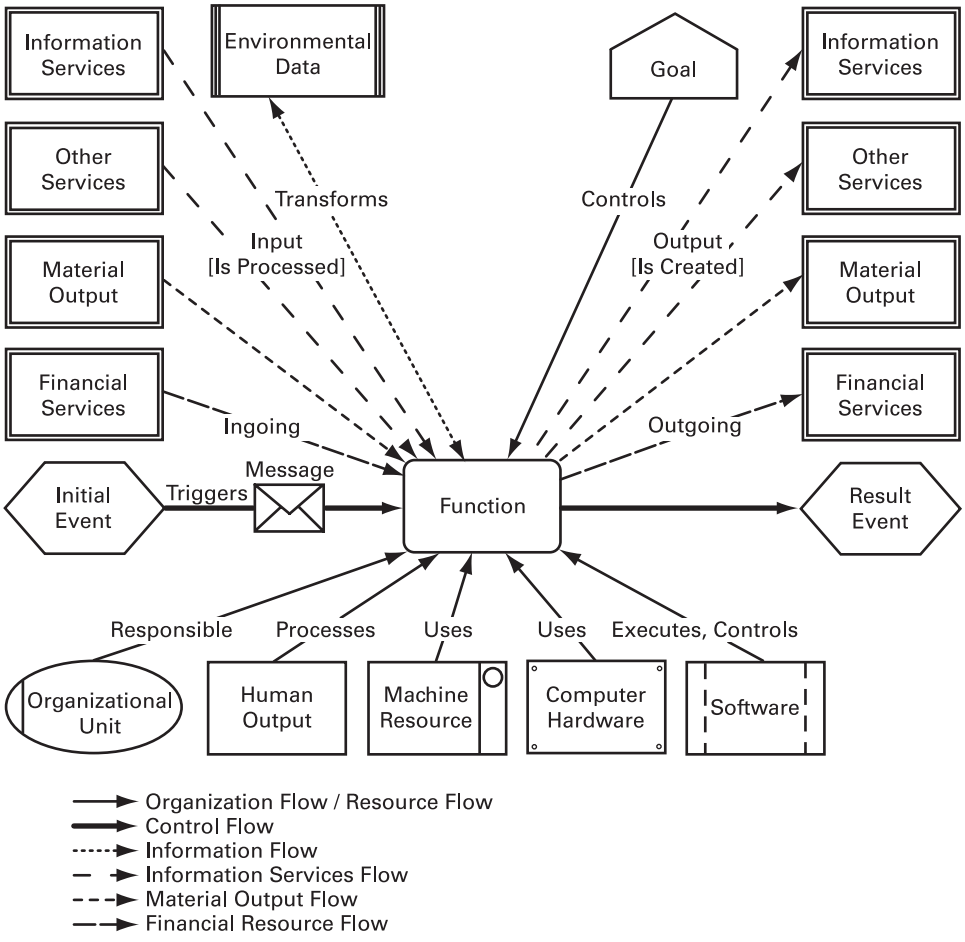


Figure 2.4
A meta-metamodel representing the interactions among the different perspectives in the ARIS House. Adapted from (Scheer 1998)

mechanisms. Microsoft has therefore decided to standardize all metadata schemas (information models) within the context of a predefined metamodel of Rational's Unified Modeling Language. An excerpt of this metamodel—a class diagram that shows the basic structure of UML class diagrams—is shown in figure 2.5.

Within this framework, MSR also supports a number of specific repository information models. In addition, the application-independent kernel of MSR offers features such as fine-grained version control.

As an example of information models, figure 2.6 shows an excerpt from the Open Information Model (OIM) (Metadata Coalition 1999), which is intended to support

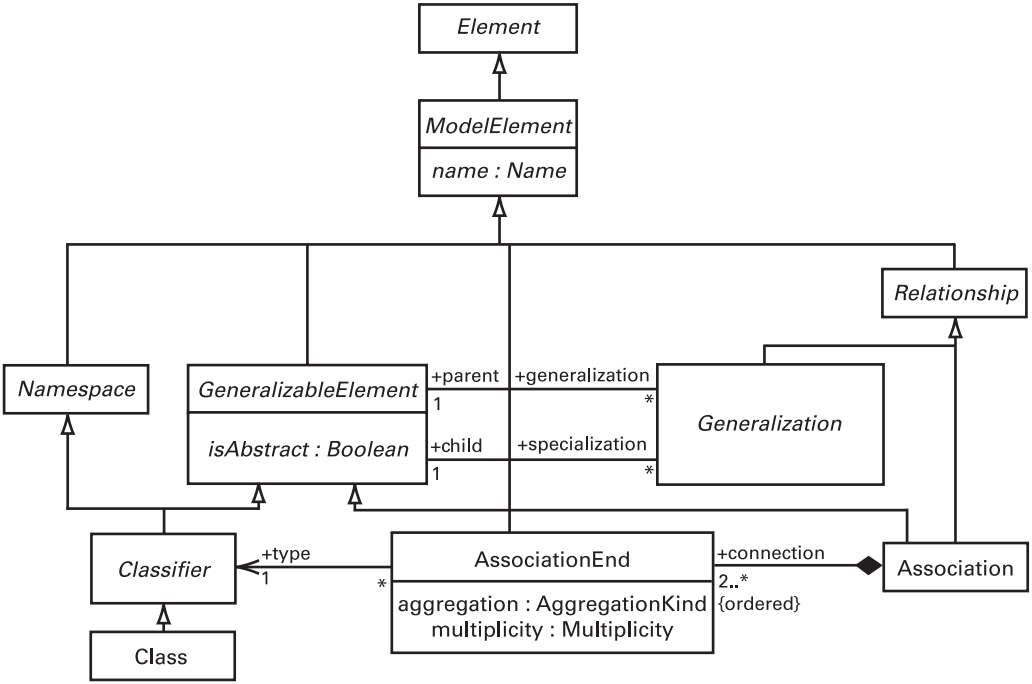


Figure 2.5
UML metamodel of UML class diagrams

life-cycle-wide tool interoperability. OIM uses UML both as a modeling language and as the basis for its core model. OIM is divided into submodels, or *packages*, that extend UML in order to address different areas of information management.

For instance, the *Data Transformations Elements* package covers basic relational-to-relational model transformations. A *transformation* maps a set of source objects to a set of target objects, both represented by a *transformable object set* (typically sources and targets are relational columns or whole tables). A transformation has a *function expression* property to provide a description of the executed code/script. A *transformation task* describes a set of transformations that must be executed together—a logical unit of work. A *transformation step* executes a single task and is used to coordinate the flow of control between tasks. A *step precedence* is a logical connector for steps: A step can be characterized by its *preceding* and *succeeding* step precedence instance. A *transformation package* is the unit of storage for transformations and consists of a set of steps, packages, and other objects. *Package executions* express the concept of data lineage, covering the instances of the *step executions*.

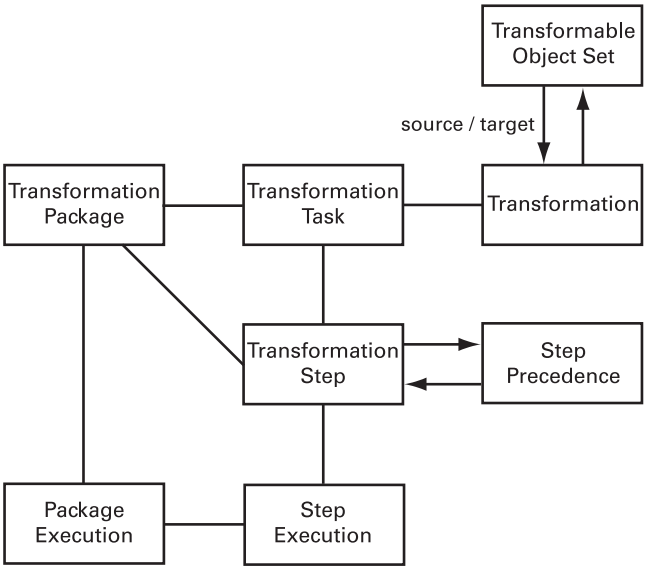


Figure 2.6
The MSR metamodel for Data Transformation Element

2.4.3 Declarative Method Engineering: MetaEdit+

MetaEdit+ (Kelly, Lyytinen, and Rossi 1996; Kelly 1998; Rossi 1998) employs a graph-object-property-relationship-role meta-metamodel to effectively specify, manage, and integrate graph-oriented design methods. MetaEdit+ was developed specifically for rapid development and deployment of *any* graphically oriented design method. To achieve this goal it offers a powerful modeling environment in which to *declaratively* model nearly any type of ontology (flat, hierarchical, integrity, and existential constraints) and map it to any type of notation that can be defined in its graphical symbol editor. It also offers default mappings to matrix and textual representations and thereby enables users to input and output ontological constructs with any type of notation. For rapid deployment it also provides a large set of repository functions that allow extensive reuse of existing metamodels and their components.

MetaEdit+ can run either in a single-user workstation environment, or simultaneously on many workstation clients connected by a network to a server (see figure 2.7). In the latter configuration, each client has a running instance of MetaEdit+, including all its tools and the MetaEngine. The MetaEngine takes care of all issues involved in communicating between the server and tools. Tools communicate with one another only through the MetaEngine and thereby through the shared GOPRR data in the repository.

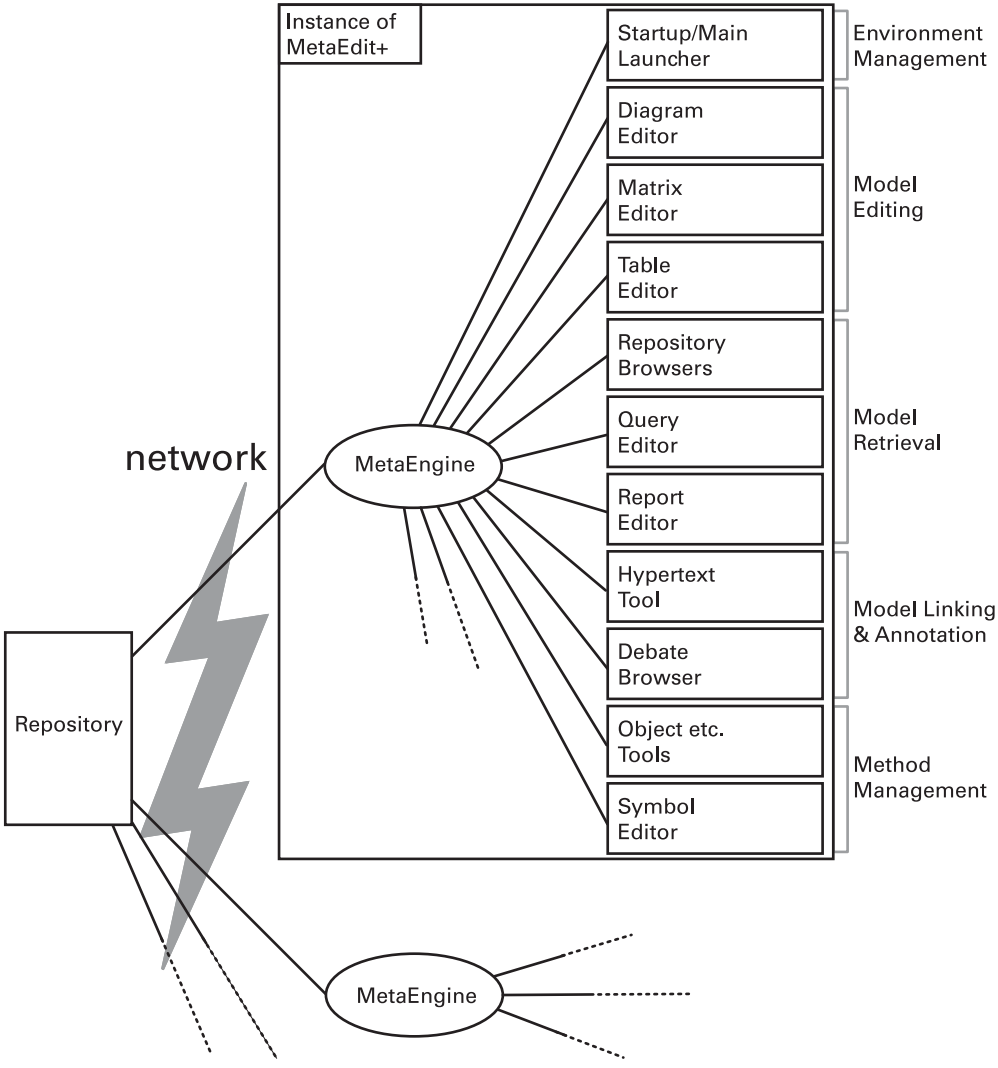


Figure 2.7
Architecture of MetaEdit+

The MetaEdit+ server forms the repository holding all the data contained in models about information systems, and also in the metamodel(s) about these models, in addition to user and locking information. The MetaEdit+ repository includes a *method specification base* containing all the method specifications as instances of the GOPRR meta-metamodel concepts; a *symbol specification base* containing all symbols needed to represent objects, relationships, and roles; a *report specification base* containing all report and other output specifications; a *model information base* containing all the conceptual instances of the GOPRR metamodels; a *representation information base* containing all information (such as spatial coordinates or size) needed to represent conceptual instances in different tools; and a *user information base* containing all information related to various users, such as their passwords or access rights, or what locks on data objects or services they currently hold.

GOPRR objects are typically discrete shapes in diagrams that can have properties (strings, numbers, etc., represented in text boxes within the shapes). Relationships can also have properties and form the hub of connections between objects, whereas roles (also with properties) form the spokes of connections and define the manner in which objects participate in specific relationships. Roles are normally represented as directed lines or arcs in the graphical models; relationships may or may not have a symbol at the hub or center of the roles. MetaEdit+ also allows multiple ontological loadings of constructs in the sense that the same construct can be an object in one graph, whereas in another graph, it can be viewed as a property. These constructs are "packaged" into hierarchically organized graphs, again with their own properties. Graphs can be included in parent graphs or attached to objects, roles, or relationships therein. MetaEdit+ allows various types of representations of each graph, in particular, graphical diagrams, textual tables, and matrix presentations.

Each graph type forms a single method; multiple methods can be defined into interconnected methodologies. MetaEdit+ offers means of defining connections between and integrity constraints on different methods: which graph elements can have subgraphs, update dependencies, value- versus name-based sharing of data etc. The whole structure is object-oriented, allowing generalization, specialization, and polymorphism in the sense that the same design items can play different roles in different notations. An example of a GOPRR specification of a data flow diagram is presented in figure 2.8.

The main purpose of MetaEdit+ is the fast generation of specialized method variants for a large variety of engineering tasks and the subsequent integration and management of designs created by multiple method variants. Metamodels can be specified using a set of method specification and management tools, which allow fast and disciplined specification and management of each part and component of a GOPRR metamodel instance. The tools are defined to encourage extensive reusabil-

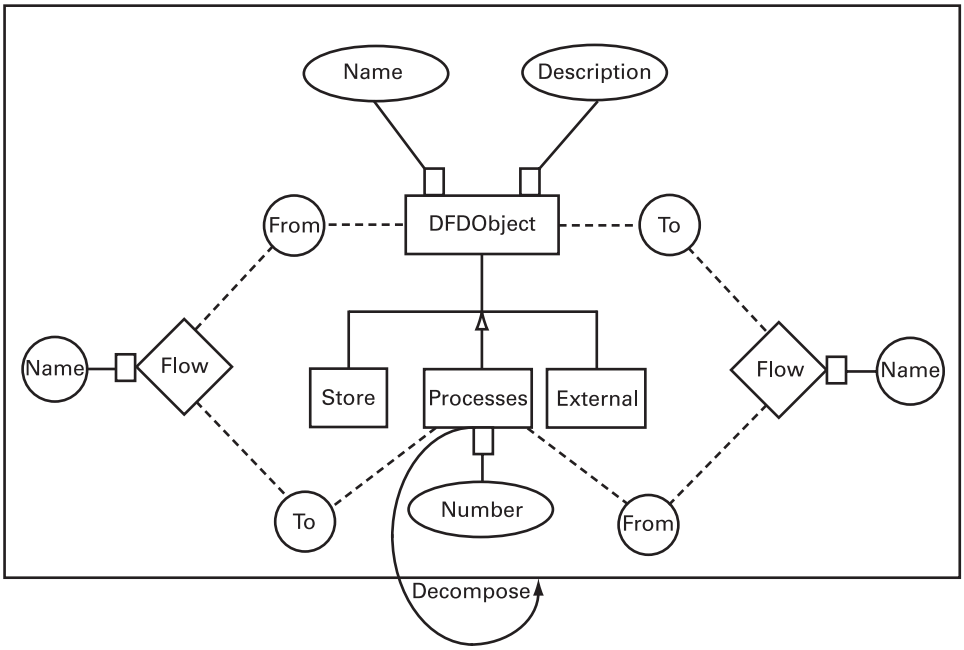


Figure 2.8
A GOPRR model of a data flow diagram

ity in that all existing metamodel specifications are available during the method specification step. Figure 2.9 shows an example of a graph tool, which the method engineer uses to put together a GOPRR graph type "package" and thereby define a method. A large variety of tools for model editing (in diagram, matrix, or table notations), model retrieval, model linkage and annotation, method management, and environment management are offered, as shown in figure 2.7. MetaEdit+ offers a scripting and report definition language to specify transformation of models into all manner of textual output: code, test scripts, documentation, configuration information, XML export of model information, etc.

MetaEdit+ is a versatile and flexible tool that is specifically dedicated to domain-based metamodeling and associated software development tasks. It has been used in many commercial projects, including design and full code generation for mobile phones based on extensive domain models, managing security and privacy policies in various types of companies, and developing business models and software solutions for insurance companies. These and other software engineering applications of MetaEdit+ demonstrate the practical usefulness of this graph-based metamodeling and integration approach that focuses on extensive domain modeling.

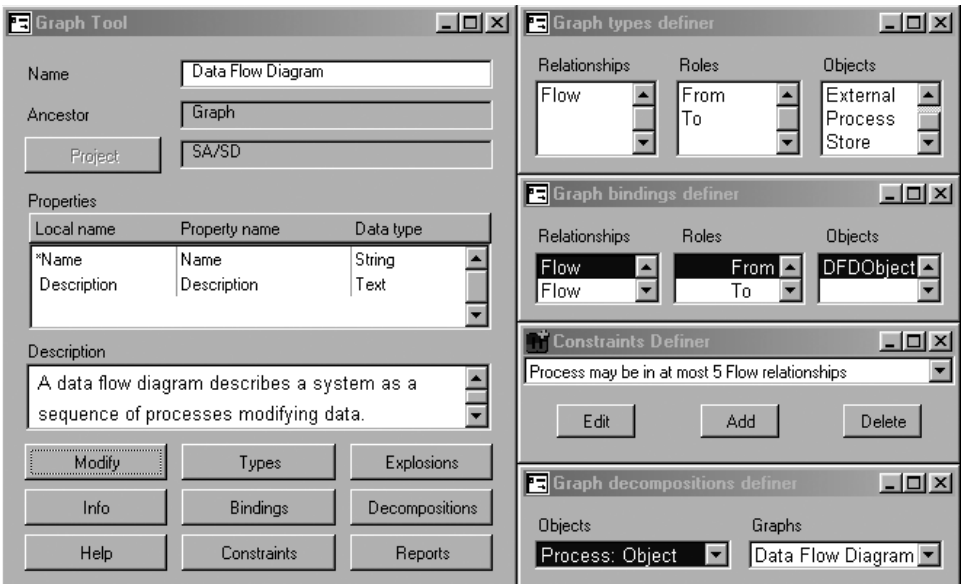


Figure 2.9
Part of the DFD metamodel in a graph tool in MetaEdit+

2.4.4 Descriptive and Integrative Metamodeling: Telos and ConceptBase

In contrast to MetaEdit+, the formal foundation of ConceptBase is not graph theory, but the Datalog formalism underlying deductive relational databases. ConceptBase was originally developed as a repository for life-cycle-wide metadata management in information systems engineering (Jarke and Rose 1988). Its formal basis was a version of the Telos metamodeling language (Mylopoulos et al. 1990) reaxiomatized in terms of Datalog with dynamically stratified negation (Jeusfeld 1992). This enables all the results on query optimization, integrity checking, and incremental view maintenance developed in the logic and object database communities to be reused (Jarke et al. 1995).

On the other hand, Telos itself is an abstraction of the pioneering formalization of the widely used structured methods by Greenspan (1984). These methods (and this has not changed in their object-oriented successors, such as UML) offer multiple modeling viewpoints, perspectives, or contexts (Motschnig-Pitrik 1995; Theodorakis et al. 2002). Managing the relationships among these viewpoints has been a central design issue in ConceptBase and its applications. The key feature Telos provides for this purpose is an unlimited instantiation hierarchy with rules and constraints for defining formal semantics across multiple levels (*metaformulas*). This hierarchy allows the full range of data, metadata, metamodels, meta-metamodels, etc., to be

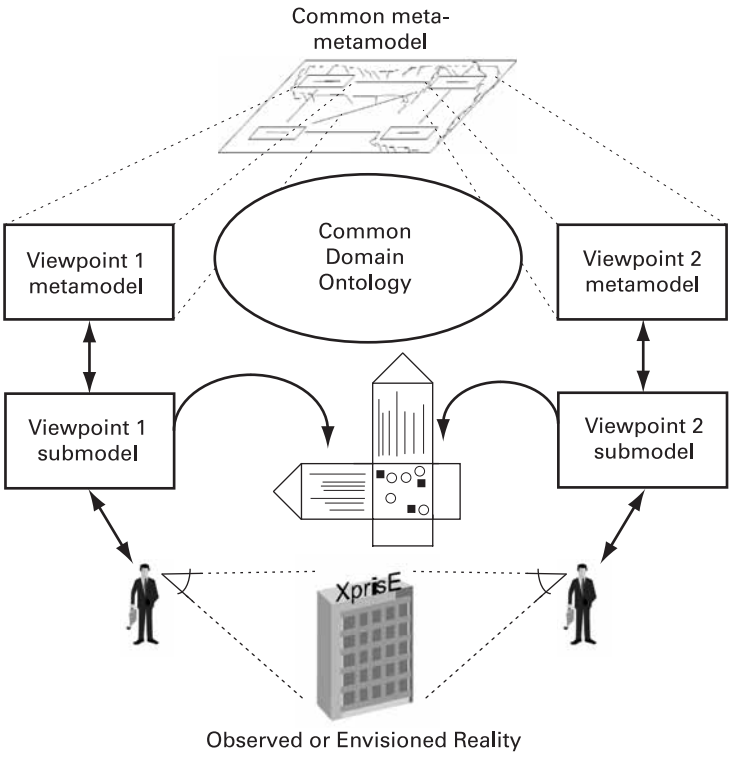


Figure 2.10
Four-level information integration with ConceptBase

managed with full querying, deduction, and integrity-checking facilities within a single repository.

The viewpoint resolution strategy shown in figure 2.10 (Nissen and Jarke 1999) focuses on the cooperative analysis of an observed or envisioned reality from multiple interrelated viewpoints. In contrast to traditional design methods that aim at orthogonality of modeling concepts, it emphasizes judicious use of viewpoint overlaps and conflicts at all levels of instantiation for quality control and knowledge elicitation.

A shared meta-metamodel provides a small core ontology of the domain of modeling similar to the language constructs used in engineering product and process modeling standard approaches such as STEP. The difference here is that our meta-metamodel comes with a fairly rich definition of meta-metaconcept semantics through metaformulas that constrain the relationships of objects within and across metamodels, models, or even data; the efficient optimization of these metaformulas constitutes a key advance in the ConceptBase implementation (Jeusfeld 1992).

Relationships between metamodels (i.e., between the constructs of different modeling formalisms used for representing heterogeneous viewpoints) were originally managed indirectly in ConceptBase, by defining each modeling construct as an instance of a specific meta-metamodel concept and then automatically specializing the associated metaformulas. In complex domains with many different modeling formalisms, this leaves too many options for inter-viewpoint constraints. The definition of more elaborate domain ontologies to which the viewpoint constructs can be related is a fashionable solution.

In a specific modeling process, further specialization of the inter-viewpoint analysis can be derived from the metaformulas. But again, this requires at least the identification and documentation of the model objects in the different viewpoints that refer to the same phenomena in reality. Thus, as figure 2.8 shows, the models need to be related not only through the shared meta-metamodel, but also by a shared grounding in reality. The resulting relationships can be documented using practice-proven matrix-based representations such as the "house of quality" from quality function deployment (Hauser and Clausing 1988). The foregoing approach has proven quite useful in applications such as business process analysis under varying theories of what constitutes good or bad business practice (Nissen et al. 1996), cooperation process analysis (Kethers 2002), reengineering of both large-scale database schemas and application code (Jeusfeld and Johnen 1995), and structured tracing of large-scale engineering processes (Ramesh and Jarke 2001).

A particularly powerful feature of the Telos language underlying ConceptBase is that the underlying Datalog foundation has completely uniform representation of objects, such that relationships can be involved in relationships, properties can have properties, and all of these objects can be provided with formal semantics through metaformulas. We illustrate these aspects with a meta-metamodel for requirements traceability that has been abstracted from a large set of empirical studies (in Ramesh and Jarke 2001) (see figure 2.11).

In the inner kernel of this model, dependencies among product objects are created by trying to satisfy another product object (goal or requirement). However, such configurations of dependency and satisfaction relationships among product objects themselves evolve in a process that is documented in process objects. Each such process evolution step can be justified by a documented rationale that is another object. In short, this metamodel reflects much of what has been described in section 2.3 as our overall metamodeling framework (except perhaps the notational aspect).

Finally, in the outer shell of the meta-metamodel, we have to consider who actually created, or commented on, all the objects and in what kind of media sources this information has been captured, that is, what the contribution structure of the conceptual product and process models is. The role of human stakeholder communities and media usage is currently gaining momentum with the increasing bandwidth of com-

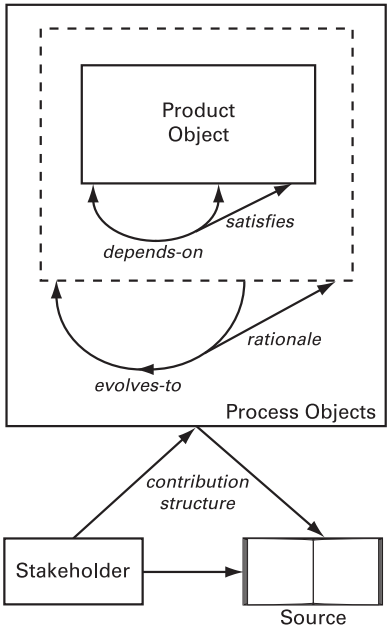


Figure 2.11
Meta-metamodel of requirements traceability

munications technology and subsequent media enrichment of cooperative work and play, as discussed in the next subsection.

2.4.5 Metamodels for Multimedia Access: MPEG-7

The development of the XML standard constitutes a major step in bringing metadata to the forefront of attention among information and communication systems developers. Like Telos, XML is in its essence a self-descriptive data description language: Each XML object contains tags that mark up the intended meaning of its components.

In basic XML, these tags are just at the grammatical level of lexical analysis, corresponding to nested brackets denoting subobjects. They allow full flexibility of self-description on the part of each object and can thus form the basis for relating almost any kinds of data objects.

Even more than in the case of Telos, XML is thus a bottom-up integration mechanism in the sense of figure 2.1. Schemas are typically considered to be views imposed on existing structures of metadata, as a mere help for the user in formulating meaningful queries in query languages such as XQuery. This approach is very suitable and flexible as long as it is humans who formulate the queries and the data

```
<Mpeg7>
   <DescriptionMetadata id="Vertigo001_vid">
      general information about the document
   </DescriptionMetadata>
       .
       .
       .
   <ContentDescription xsi:type="ContentEntityType">
      <MultimediaContent xsi:type="VideoType">
                .
                .
                .
            <VideoSegment id="seg1">
                    .
                    .
                    .
               <TextAnnotation type="User_***">
                  Beauty
                  Location
                  Setting
                  Portrait
               </TextAnnotation>
                    .
                    .
                    .
            </VideoSegment>
                .
                .
                .
      </MultimediaContent>
   </ContentDescription>
       .
       .
       .
</Mpeg7>
```

Figure 2.12
A simple MPEG-7/XML example

objects retrieved have only limited variability in terms of representation and media employed.

Recently, however, the vision of the World Wide Web has shifted toward that of a "Semantic Web" (Berners-Lee, Hendler, and Lassila 2001) in which Internet content is not just visible to humans but also processible by machines. For multimedia objects in particular, machine readability requires a rich set of standard metadata that cannot be offered by simple existing standards such as Dublin Core or RDF (Resource Description Framework) but must basically include all kinds of metadata aspects shown in figure 2.2.

MPEG-7 is a standard that has been in development since about 1999 by the Moving Pictures Expert Group to meet this challenge (Manjunath, Salembier, and Sikora

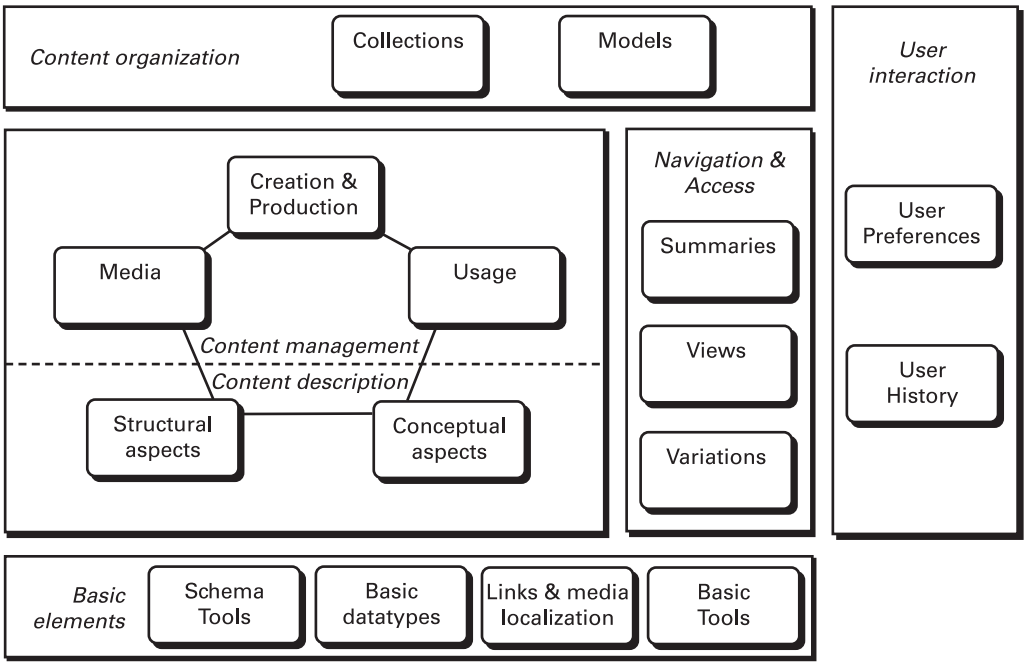


Figure 2.13
Description schemes of the MPEG-7 metadata standard

2002). In addition to the search, retrieval, and filtering of multimedia information, the editing, combination, and reorganization of media as well as the personalization of user access (Riecken 2001) are supported in a notational framework based on XML. Figure 2.12 shows a cut-out of a MPEG-7 file used in the environment described subsequently.

MPEG-7 metadata are contained in a kind of library of *multimedia description schemes*, organized into groups (see figure 2.13). *Basic elements* include, for example, data types (e.g., media time and duration) and schema tools for localization and text annotation. *Content management* forms the kernel of the multimedia description schemes; the figure shows that it addresses ontological and presentational (''structural'') aspects and also process aspects from our discussion in section 2.3; even goal aspects such as legal and financial information are covered in the *usage* scheme. *Content description* defines the structural and semantic aspects of multimedia metadata. The organizational groupings of *content organization* and *navigation|access* allow the organization of multimedia material according to different collections and views, whereas *user interaction* is responsible for handling personalization and traceability.

Figure 2.14
MECCA: An MPEG-7-based video triage system

As an example of MPEG-7 metadata management, figure 2.14 shows the Movie Classification and Categorization Application (MECCA), a high-level semantic annotation tool for multimedia artifacts (Klamma, Spaniol, and Jarke 2005). MECCA serves as a multimedia environment for online video triage and collaborative ontology creation, which is a discursive and multistage process. MECCA is being used in the cinematic sciences by users having diverse educational backgrounds, like cinematic science, history of art, or graphical design. This community of practice brings together users at various stages of their professional careers, such as full professors, research assistants, and students. Members of the community have different interests and points of view as a result of their educational backgrounds. In MECCA, users first check already existing multimedia content. In addition, users can add content compatible with MPEG-7. The next step is performed by gradually annotating and classifying the data. Each user classification schema is kept in a separate MPEG-7 file. In order to retain the semantics of a multimedia file, individual annotations and

classifications are made possible by allowing redundant, overlapping, or even divergent views. These personal collections can be distributed and discussed among other community members.

MECCA is based on the constructivist learning environment "Berliner sehen" (Fendt 2001). Scientists are stimulated to freely explore multimedia content, led by the high-level semantics of the MPEG-7 content description, and to share their experiences in collections with others via the Web. The front end of our video triage application (see figure 2.14) is an enhancement of its predecessor, the Virtual Entrepreneurship Lab (VEL) (Klamma et al. 2002). The buttons in the column on the right-hand side of the figure reflect the domain ontology externalized by the scientists in an ongoing discourse that organizes the content in the MPEG-7 model. Selecting a combination of buttons instigates a similarity search in the underlying XML database that returns the video thumbnails in the gallery shown in the left part of the figure according to prestated user preferences. In the middle of this gallery is an MPEG-7-compliant multimedia player/editor that serves different audio/video/image/virtual reality content. The environment displays the underlying structure with respect to the cardinality of categories and components in the classification schema for media content being used. To serve different classification systems, further presentation styles from one-dimensional buttons to trees (as well as a combination of both) can be selected adaptively. An additional full-text search engine allows a comprehensive search on user collections or on the original content. Finally, the lower left part of the interface allows the user to reorganize content in personalized collections of multimedia objects and to annotate them with additional goals or rationales.

As a result of discursive knowledge creation processes in communities of scientists from the humanities, an interest has arisen in exploring distributed classification processes. About a year ago, the MECCA environment was introduced to our colleagues in the humanities. Starting with meetings of six to eight community members, the community of scientists initially defined a common classification schema for multimedia content on a drawing table. Members wanted to define a common vocabulary, but some terms offered for inclusion in this common vocabulary were criticized, since the disciplines of community members cover a wide range. Hence, the community rejected some terms for inclusion, since their interpretation might have been misleading. Some special terms were included in the common classification scheme to allow specialists to classify the multimedia content in detail. These terms do not conflict with the intuitive understanding of others, but because of the degree of their specialization into a subsection of the cinematic sciences, these terms are rarely used. Hence, researchers hope that a computer-mediated system could manage conflicts of understanding more accurately.

The use of MPEG-7 has been crucial for the development of MECCA. MPEG-7 allows users to express high-level multimedia semantics in a collaborative knowledge creation process. In addition, heterogeneous information can be contextualized, since MPEG-7 gives users the ability to manage content in diverse digital media formats.

2.5 Concluding Remarks

Metadata management and metamodeling has, for a long time, been considered a rather boring bookkeeping exercise. The recent increase in interest in this field is only partially reflected in the examples discussed in section 2.4.

This increase in interest is due first to the flexibility and variation in the methods required for efficient systems engineering in a globalized competitive setting of ubiquitous information technology. Second, there is a major drive for better information integration and more rapid change. Third, the increasing media richness and ubiquity of contexts for computer usage and computer-mediated communications have contributed to the need for more rigorous metadata handling. In particular, the Semantic Web prediction that future Internet traffic will be generated more by machines talking to machines than simply by human-computer interaction will require even further progress in this area.

Acknowledgments

This work was supported in part by Deutsche Forschungsgemeinschaft in project PRIME and in SFB 427 and 476, and by the European Community under IST project SEWASIE.

References

Aalto, J.-M. 1993. "Experiences on Applying OMT to Large Scale Systems." In *Proceedings of the Seminar on Conceptual Modeling and Object-Oriented Programming*, ed. A. Lehtola and J. Jokiniemi, 39–47. Helsinki: Finnish Artificial Intelligence Society.

Abiteboul, S., P. Buneman, and D. Suciu. 2000. *Data on the Web—From Relations to Semistructured Data and XML*. San Francisco: Morgan Kaufmann.

Alford, M. 1992. "Strengthening the Systems/Software Engineering Interface for Real Time Systems." In *Proceedings of the Second International Symposium of the National Council on Systems Engineering (NCOSE)*, 411–418. Sunnyvale, CA: NCOSE.

Armenise, P., S. Bandinelli, C. Ghezzi, and A. Morzenti. 1993. "A Survey and Assessment of Software Process Representation Formalisms." *International Journal of Software Engineering and Knowledge Engineering* 3, no. 3: 410–426.

Avaro, O., and P. Salembier. 2001. "MPEG-7 Systems: Overview." *IEEE Transactions on Circuits and Systems for Video Technology* 11, no. 6: 760–764.

Bandinelli, S., A. Fuggetta, and C. Ghezzi. 1993. "Software Process Model Evolution in the SPADE Environment." *IEEE Transactions on Software Engineering* 19, no. 12: 1128–1144.

Basili, V. R., and H. D. Rombach. 1988. "The TAME Project: Towards Improvement-Oriented Software Environments." *IEEE Transactions on Software Engineering* 14, no. 6: 758–773.

Baumeister, M. 1996. "Attribute Grouping: Emulating Meta Models without Instantiation." In *Proceedings of the Third International Conference on Object-Oriented Information Systems (OOIS'96)*, ed. D. Patel, Y. Sun, and S. Patel, 169–179. London: Springer.

Bergsten, P., J. Bubenko, R. Dahl, M. Gustafsson, and L.-Å. Johansson. 1989. "RAMATIC—A CASE Shell for Implementation of Specific CASE Tools." TEMPORA Technical Report No. T6.1, Swedish Institute for System Development (SISU), Stockholm.

Berners-Lee, T., J. Hendler, and O. Lassila. 2001. "The Semantic Web." *Scientific American* (May). 284, no. 5: 34–44.

Bernstein, P. A. 2001. "Generic Model Management—A Database Infrastructure for Schema Management." In *Proceedings of the Ninth International Conference on Cooperative Information Systems (CoopIS'01)* (Lecture Notes in Computer Science 2172), ed. C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, 1–6. New York: Springer.

Bernstein, P. A., T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, and D. Shutt. 1999. "Microsoft Repository Version 2 and the Open Information Model." *Information Systems* 24, no. 2: 71–98.

Bernstein, P. A., and U. Dayal. 1994. "An Overview of Repository Technology." In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, ed. J. Bocca, M. Jarke, and C. Zanioli, 705–713. San Francisco: Morgan Kaufmann.

Boland, R., and R. Tenkasi. 1995. "Perspective Making and Perspective Taking in Communities of Knowing." *Organization Science* 6, no. 4: 350–372.

Boloix, G., P. G. Sorenson, and J. P. Tremblay. 1991. "On Transformations Using a Metasystem Approach to Software Development." Technical report, Department of Computing Science, University of Alberta, Edmonton.

Booch, G. 1994. *Object-Oriented Analysis and Design with Applications*. Redwood City, CA: Benjamin Cummings.

Booch, G., J. Rumbaugh, and I. Jacobson. 1998. *Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley.

Bowman, C. M., P. Danzig, D. Hardy, U. Manber, and M. Schwartz. 1995. "The Harvest Information Discovery and Access System." *Computer Networks and ISDN Systems* 28, no. 1–2: 119–125.

Brodie, M. L. 1997. "Silver Bullet Shy on Legacy Mountain: When Neat Technology Just Doesn't Work or . . . Miracles to Save the Realm: Faustian Bargains or Noble Pursuits?" In *Proceedings of the 16th International Conference on Conceptual Modeling (ER'97)* (Lecture Notes in Computer Science 1331), ed. D. Embley and R. Goldstein, 183. New York: Springer.

Brodie, M. L., J. Mylopoulos, and J. W. Schmidt. 1984. *On Conceptual Modeling*. New York: Springer-Verlag.

Calvanese, D., G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. 2001. "Data Integration in Data Warehousing." *International Journal of Cooperative Information Systems* 10, no. 3: 237–272.

Carlile, P. R. 2002. "A Pragmatic View of Knowledge and Boundaries: Boundary Objects in New Product Development." *Organization Science* 13, no. 4: 442–455.

Castro, J., M. Kolp, and J. Mylopoulos. 2002. "Towards Requirements-Driven Information Systems Engineering: The Tropos Project." *Information Systems* 27, no. 6: 365–390.

Catarci, T., and M. Lenzerini. 1993. "Interschema Knowledge in Cooperative Information Systems." *Proceedings of the International Conference on Intelligent and Cooperative Information Systems (CoopIS'93)*, ed. G. Schlageter, M. Huhns, M. Papazoglou, 55–62. Los Alamitos, CA: IEEE Computer Society.

CDIF (CASE Data Interchange Format). 1994. "Framework for Modeling and Extensibility." Technical report EIA/IS-107, Electronic Industries Association, Arlington, VA.

Checkland, P. B. 1981. *Systems Thinking, Systems Practice.* Chichester, England: Wiley.

Chen, M., and R. J. Norman. 1992. "A Framework for Integrated CASE." *IEEE Software* 9, no. 2: 18–22.

Chiba, S., and T. Masuda. 1993. "Designing an Extensible Distributed Language with a Meta-level Architecture." In *Proceedings of the Seventh European Conference on Object-Oriented Programming (ECOOP'93)* (Lecture Notes in Computer Science 707), ed. O. Nierstrasz, 483–502. Berlin: Springer.

Coleman, D., P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. 1994. *Object-Oriented Development—The Fusion Method*. Englewood Cliffs, NJ: Prentice Hall.

Constantopoulos, P., M. Jarke, J. Mylopoulos, and Y. Vassiliou. 1995. "The Software Information Base: A Server for Reuse." *VLDB Journal* 5, no. 1: 1–42.

Curtis, B., M. I. Kellner, and J. Over. 1992. "Process Modeling." *Communications of the ACM* 35, no. 9: 75–90.

Cybulski, J. L., and K. Reed. 1992. "A Hypertext Based Software Engineering Environment." *IEEE Software* 9, no. 2: 62–68.

Dardenne, A., A. van Lamsweerde, and S. Fickas. 1993. "Goal-directed Requirements Acquisition." *Science of Computer Programming* 20: 3–50.

Davis, G.-B. 1982. "Strategies for Information Requirements Determination." *IBM Systems Journal* 21, no. 1: 4–30.

Davis, R., and D. Lenat. 1982. *Knowledge-Based Systems in Artificial Intelligence*. New York: McGraw-Hill.

Deutsch, P., and A. Emtage. 1994. "Publishing Information on the Internet via Anonymous FTP." Available at ⟨http://www.ifla.org/documents/libraries/cataloging/metadata/iafa.txt⟩.

Farquhar, A., Fikes, R., and Rice, J. 1997. "The Ontolingua Server: A Tool for Collaborative Ontology Construction." *International Journal of Human-Computer Studies* 46, no. 6: 707–727.

Fendt, K. 2001. "Contextualizing Content." In *Languages across the Curriculum*, ed. M. Knecht and K. von Hammerstein, 201–223. Columbus, OH: National East Asian Language Center.

Finkelstein, A., J. Kramer, and B. Nuseibeh. 1994. *Software Process Modelling Technology*. New York: Wiley Research Science.

Fitzgerald, B. 1995. "The Use of Systems Development Methods: A Survey." Economic and Social Research Council research and discussion paper, University College, Cork, Ireland.

Fuxman, A., M. Pistore, J. Mylopoulos, and P. Traverso. 2001. "Model-Checking Early Requirements in Tropos." In *Proceedings of the Fifth IEEE Symposium on Requirements Engineering (RE'01)*, 174–181. Los Alamitos, CA: IEEE Computer Society.

Gaines, B. R., D. H. Norrie, A. Z. Lapsley, and M. L. G. Shaw. 1996. "Knowledge Management for Distributed Enterprises." In *Proceedings of the Tenth Knowledge Acquisition Workshop*, ed. B. Gaines and M. Musen. Available at ⟨http://ksi.cpsc.ucalgary.ca/KAW/KAW96/KAW96Proc.html⟩.

Gans, G., M. Jarke, S. Kethers, G. Lakemeyer, L. Ellrich, C. Funken, and M. Meister. 2001. "Requirements Modeling for Organization Networks: A (Dis)Trust-Based Approach." In *Proceedings of the Fifth IEEE Symposium on Requirements Engineering (RE'01)*, 154–163. Los Alamitos, CA: IEEE Computer Society.

Ghezzi, C., and B. Nuseibeh, eds. 1998/1999. "Managing Inconsistency in Software Development." Special section, *IEEE Transactions on Software Engineering* 24, no. 11: 906–1001 and 25, no. 6: 782–869.

Goble, C. A., R. Stevens, G. Ng, S. Bechhofer, N. W. Paton, P. G. Baker, M. Peim, and A. Brass. 2001. "Transparent Access to Multiple Bioinformatics Information Sources." *IBM Systems Journal* 40, no. 2: 532–551.

Goh, C. H., S. Bressane, S. E. Madnick, and M. D. Siegel. 1999. "Context Interchange: New Features and Formalisms for the Intelligent Integration of Information." *ACM Transactions on Information Systems* 17, no. 3: 270–293.

Gomez-Peres, A., and O. Corcho. 2002. "Ontology Languages for the Semantic Web." *IEEE Intelligent Systems* 17, no. 1: 54–60.

Greenspan, S. 1984. "Requirements Modeling: A Knowledge Representation Approach to Requirements Definition." Ph.D. diss., Department of Computer Science, University of Toronto, Toronto.

Gross, T., and M. Specht. 2001. "Awareness in Context-Aware Information Systems." In *Proceedings of Mensch & Computer 01*, ed. H. Oberquelle, R. Oppermann, and J. Krause, 173–182. Stuttgart: Teubner.

Gruber, T. R. 1993. "A Translation Approach to Portable Ontology Specifications." *Knowledge Acquisition* 5, no. 2: 199–220.

Guarini, N., and C. Welty. 2002. "Conceptual Modeling and Ontological Analysis." Keynote talk presented at *the Fourteenth Conference on Advanced Information System Engineering (CAiSE'02)*. Toronto. Available at ⟨http://www.cs.toronto.edu/caise02/⟩.

Harmsen, F., S. Brinkkemper, and H. Oei. 1994. "Situational Method Engineering for Information System Projects." In *Proceedings of the Internationl Federation for Information Processing Working Group 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies (CRIS'94)*, ed. T. W. Olle and A. A. Verrijn-Stuart, 169–194. Amsterdam: North-Holland.

Harmsen, F., I. Lubbers, and G. Wijers. 1995. "Success-Driven Selection of Fragments for Situational Methods: The S3 Model." In *Proceedings of the Second International Workshop on Requirements Engineering: Foundations of Software Quality*, ed. K. Pohl and P. Peters, 104–115. Aachen, Germany: Augustinus.

Harmsen, F., and M. Saeki. 1996. "Comparison of Four Method Engineering Languages." In *Method Engineering: Principles of Method Construction and Tool Support*, ed. S. Brinkkemper, K. Lyytinen, and R. J. Welke, 209–231. New York: Springer.

Hauser, J. R., and D. Clausing. 1988. "The House of Quality." *Harvard Business Review* 66, no. 5: 63–73.

Heineman, G. T., J. E. Botsford, G. Caldiera, G. E. Kaiser, M. I. Kellner, and N. H. Madhavji. 1994. "Emerging Technologies That Support a Software Process Life Cycle." *IBM Systems Journal* 33, no. 3: 501–529.

Hernandez, J. 1997. *The SAP R|3 Handbook*. New York: McGraw-Hill.

Hill, P. M., and J. W. Lloyd. 1989. "Analysis of Metaprograms." In *Metaprogramming in Logic Programming*, ed. H. D. Abramson and M. H. Rogers, 23–52. Cambridge, MA: MIT Press.

Hirschheim, R., H. Klein, and K. Lyytinen. 1995. *Information Systems Development—Conceptual and Philosophical Foundations*. New York: Cambridge University Press.

Hong, S., S. Brinkkemper, and F. Harmsen. 1995. "Object-Oriented Method Components for Situation-Specific IS Development." In *Proceedings of the Fifth Annual Workshop on Information Technologies and Systems*, ed. S. Ram and M. Jarke, 164–173. Aachener Informatik Berichte technical report 1995-15, RWTH Aachen, Aachen, Germany.

Horrocks, I. 2002. "DAML+OIL—A Reasonable Web Ontology Language." In *Proceedings of the Eighth International Conference on Extending Data Base Technology (EDBT)* (Lecture Notes in Computer Science 2287), ed. C. S. Jensen, K. G. Jeffery, J. Pokorny, S. Saltenis, E. Bertino, K. Böhm, and M. Jarke, 2–14. New York: Springer.

Huhns, M. N., N. Jacobs, T. Ksiezyk, W.-M. Shen, M. P. Singh, and P. E. Cannata. 1993. "Integrating Enterprise Information Models in Carnot." In *Proceedings of the First International Conference on Intelligent and Cooperative Information Systems*, 32–42. Los Alamitos, CA: IEEE Computer Society.

Iivari, J., and P. Kerola. 1983. "A Sociocybernetic Framework for the Feature Analysis of Information Systems Development Methodologies." In *Information Systems Development Methodologies: A Feature Analysis*, ed. T. W. Olle, H. G. Sol, and C. J. Tully, 87–139. Amsterdam: North-Holland.

ISDOS (Information System Design and Optimization System). 1981. "An Introduction to the System Encyclopedia Manager." Reference no. 81, SEM-0338-1, ISDOS Project, Department of Industrial and Operations Engineering, University of Michigan, Ann Arbor.

ISO (International Organization for Standardization)/IEC (International Electrotechnical Commission). 1990. "Information Technology—Information Resource Dictionary System (IRDS)—Framework." ISO/IEC International Standard 10027. Geneva: ISO.

Jaccheri, M. L., and R. Conradi. 1993. "Techniques for Process Model Evolution in EPOS." *IEEE Transactions on Software Engineering* 19, no. 12: 1145–1156.

Jarke, M. 1992. "Strategies for Integrating CASE Environments." *IEEE Software* 9, no. 2: 54–61.

Jarke, M., ed. 1993. *Database Application Engineering with DAIDA*. Heidelberg, Germany: Springer-Verlag.

Jarke, M., R. Gallersdörfer, M. A. Jeusfeld, M. Staudt, and S. Eherer. 1995. "ConceptBase: A Deductive Object Base for Meta Data Management." *Journal of Intelligent Information Systems* 4, no. 2: 167–192.

Jarke, M., M. A. Jeusfeld, C. Quix, and P. Vassiliadis. 1999. "Architecture and Quality in Data Warehouses—An Extended Repository Approach." *Information Systems* 24, no. 3: 229–253.

Jarke, M., K. Pohl, C. Rolland, and J.-R. Schmitt. 1994. "Experience-Based Method Evaluation and Improvement: A Process Modeling Approach." In *Proceedings of the International Federation for Information Processing Working Group 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies (CRIS'94)*, ed. T. W. Olle and A. A. Verrijn-Stuart, 1–27. Amsterdam: North-Holland.

Jarke, M., K. Pohl, K. Weidenhaupt, K. Lyytinen, P. Marttiin, J.-P. Tolvanen, and M. Papazoglou. 1998. "Meta Modeling: A Formal Basis for Interoperability and Adaptability." In *Information Systems Interoperability*, ed. B. Krämer, M. Papazoglou, and H.-W. Schmidt, 229–263. New York: Wiley Research Science.

Jarke, M., and T. Rose. 1988. "Managing the Evolution of Information Systems." In *Proceedings of the ACM International Conference on Management of Data (ACM SIGMOD'88)*, ed. H. Boral and P.-A. Larson, 303–311. New York: ACM Press.

Jeusfeld, M. A. 1992. *Change Control in Deductive Object Bases* [in German]. St. Augustin, Germany: Infix.

Jeusfeld, M. A., and U. Johnen. 1995. "An Executable Meta Model for Re-engineering Database Schemas." *International Journal of Cooperative Information Systems* 4, nos. 2–3: 237–258.

Jeusfeld, M. A., and M. Papazoglou. 1999. "Information Brokering." In *Information Systems Interoperability*, ed. B. Krämer, M. Papazoglou, and H.-W. Schmidt, 265–302. New York: Wiley Research Science.

Johnson, R., and M. Palaniappan. 1993. "MetaFlex: A Flexible Metaclass Generator." In *Proceedings of the Seventh European Conference on Object-Oriented Programming (ECOOP'93)* (Lecture Notes in Computer Science 707), ed. O. Nierstrasz, 503–528. Berlin: Springer-Verlag.

Kelly, S. 1998. "Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+." Ph.D. diss., Department of Computer Science and Information Systems, University of Jyväskylä.

Kelly, S., K. Lyytinen, and M. Rossi. 1996. "METAEDIT+—A Fully Configurable Multi-user and Multi-tool CASE and CAME Environment." In *Advanced Information Systems Engineering* (Lecture Notes in Computer Science 1080), ed. P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, 1–21. New York: Springer-Verlag.

Kethers, S. 2002. "Capturing, Formalising, and Analysing Cooperation Processes—A Case Study." In *Proceedings of the Tenth European Conference on Information Systems (ECIS 2002)*, 1113–1123. Available at ⟨http://csrc.lse.ac.uk/asp/aspecis/20020138.pdf⟩.

Kiczales, G., J. des Rivieres, and D. Bobrow. 1991. *The Art of the Metaobject Protocol*. Cambridge, MA: MIT Press.

Klamma, R., E. Hollender, M. Jarke, P. Moog, and V. Wulf. 2002. "Vigils in a Wilderness of Knowledge: Metadata in Learning Environments." In *Proceedings of the IEEE International Conference on Advanced Learning Technologies (ICALT 2002)*, ed. P. Kommers, V. Petrushin, Kinshuk, and I. Galeev, 519–524. Palmerston, NZ: IEEE Learning Task Force.

Klamma, R., M. Spaniol, and M. Jarke. 2005. "MECCA: Multimedia Capturing of Collaborative Scientific Discourses about Movies." *Informing Science* 8: 3–38. Available at ⟨http://inform.nu/Articles/Vol8/indexV8summary.htm⟩.

Klas, W., and M. Schrefl. 1995. *Metaclasses and Their Applications*. Berlin: Springer-Verlag.

Kleene, S. 1967. *Mathematical Logic*. New York: Wiley.

Koskinen, M. 2000. "Process Metamodeling: Conceptual Foundations and Application." Ph.D. diss. and Jyväskylä Studies in Computing no. 7, Department of Computing Science and Information Systems, University of Jyväskylä, Jyväskylä, Finland.

Kotteman, J., and B. Konsynski. 1984. "Information Systems Planning and Development: Strategic Postures and Methodologies." *Journal of Management Information Systems* 1, no. 2: 45–63.

Koubarakis, M., and D. Plexousakis. 2002. "A Formal Framework for Business Process Modeling and Design." *Information Systems* 27, no. 5: 299–320.

Kowalski, R. A. 1979. "Algorithm = Logic + Control." *Communications of the ACM* 22, no. 7: 424–436.

Kremer, R. 1996. "Toward a Multi-user, Programmable Web Concept Mapping Shell to Handle Multiple Formalisms." In *Proceedings of the Tenth Knowledge Acquisition Workshop*, ed. B. Gaines and M. Musen. Available at ⟨http://ksi.cpsc.ucalgary.ca/KAW/KAW96/KAW96Proc.html⟩.

Kumar, K., and R. J. Welke. 1992. "Methodology Engineering: A Proposal for Situation-Specific Methodology Engineering." In *Challenges and Strategies for Research in Systems Development*, ed. W. W. Cotterman and J. A. Senn, 257–269. New York: Wiley.

Lagoze, C. 1996. "The Warwick Framework." *D-Lib Magazine* (July/August).

Lefering, M. 1993. "An Incremental Integration Tool between Requirements Engineering and Programming in the Large." In *Proceedings of the First International Symposium on Requirements Engineering*, 82–89. Los Alamitos, CA: IEEE Computer Society.

Lehman, M., and W. Turski. 1987. "Essential Properties of IPSEs." *ACM SIGSOFT Software Engineering Notes* 12, no. 1: 52–56.

Lonchamp, J. 1993. "A Structured Conceptual and Terminological Framework for Software Process Engineering." In *Proceedings of the Second International Conference on Software Process*, 41–53. Los Alamitos, CA: IEEE Computer Society.

Lonchamp, J. 1995. "CPCE: A Kernel for Building Flexible Collaborative Process-Centered Environments." In *Proceedings of the Seventh Conference on Software Engineering Environments*, 95–105. Los Alamitos, CA: IEEE Computer Society.

Lott, C. M. 1993. "Process and Measurement Support in SEEs." *ACM SIGSOFT Software Engineering Notes* 18, no. 4: 83–93.

Manjunath, B. S., P. Salembier, and T. Sikora, eds. 2002. *Introduction to MPEG-7*. Chichester, England: Wiley.

Marquardt, W. 1996. "Trends in Computer-Aided Process Modeling." *Computers in Chemical Engineering* 20, nos. 6–7: 591–609.

Marttiin, P., F. Harmsen, and M. Rossi. 1996. "A Functional Framework for Evaluating Method Engineering Environments: The Case of Maestro II/Decamerone and MetaEdit+." In *IFIP Working Conference on Principles of Method Construction and Tool Support*, ed. S. Brinkkemper, K. Lyytinen, and R. J. Welke, 63–86. London: Chapman and Hall.

Marttiin, P., M. Rossi, V.-P. Tahvanainen, and K. Lyytinen. 1993. "A Comparative Review of CASE Shells: A Preliminary Framework and Research Outcomes." *Information and Management* 25, no. 1: 11–31.

McChesney, I. R. 1995. "Toward a Classification Scheme for Software Process Modeling Approaches." *Information and Software Technology* 37, no. 7: 363–374.

Merbeth, G. 1991. "Maestro II—das integrierte CASE-System von Softlab." In *CASE Systeme und Werkzeuge*, ed. H. Balzert, 319–336. Mannheim, Germany: BI Wissenschaftsverlag.

Mercurio, V. J., B. F. Meyers, A. M. Nisbet, and G. Radin. 1990. "AD/Cycle Strategy and Architecture." *IBM Systems Journal* 29, no. 2: 170–188.

Metadata Coalition. 1999. Open Information Model Version 1.0. Available at ⟨www.mdcinfo.com/OIM/OIM10.html⟩.

Motschnig-Pitrik, R. 1995. "An Integrating View on the Viewing Abstraction—Contexts and Perspectives in Software Development." *Journal of Systems Integration* 5, no. 1: 23–60.

Motschnig-Pitrik, R., and J. Mylopoulos. 1992. "Classes and Instances." *International Journal of Cooperative Information Systems* 1, no. 1: 61–92.

Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis. 1990. "Telos: Representing Knowledge about Information Systems." *ACM Transactions on Information Systems* 8, no. 4: 325–362.

National Library of Medicine. 1994. *Unified Medical Language Systems*. 5th ed. Washington, DC: U.S. Department of Health and Human Services.

NATURE Team. 1996. "Defining Visions in Context: Models, Processes, and Tools for Requirements Engineering." *Information Systems* 21, no. 6: 515–547.

Necco, C. R., C. L. Gordon, and N. W. Tsai. 1987. "Systems Analysis and Design: Current Practices." *MIS Quarterly* 11, no. 4: 461–475.

Neches, R., R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. 1991. "Enabling Technology for Knowledge Sharing." *AI Magazine* 12, no. 3: 36–55.

Nissen, H. W., and M. Jarke. 1999. "Repository Support for Multi-perspective Requirements Engineering." *Information Systems* 24, no. 2: 131–158.

Nissen, H. W., M. A. Jeusfeld, M. Jarke, G. Zemanek, and H. Huber. 1996. "Managing Multiple Requirements Perspectives with Meta Models." *IEEE Software* 13, no. 2: 37–48.

Oberweis, A., G. Scherrer, and W. Stucky. 1994. "INCOME/STAR—Methodology and Tools for the Development of Distributed Information Systems." *Information Systems* 19, no. 8: 643–660.

Oei, J. L. H., and E. D. Falkenberg. 1994. "Harmonisation of Information System Modelling and Specification Techniques." In *Proceedings of the International Federation for Information Processing Working Group 8.1 Working Conference on Comparative Review of Information Systems Design Methodologies (CRIS'94)*, ed. T. W. Olle and A. A. Verrijn–Stuart, 151–168. Amsterdam: North-Holland.

Olle, T. W., J. Hagelstein, I. G. MacDonald, C. Rolland, H. G. Sol, F. J. M. Van Assche, and A. A. Verrijn-Stuart. 1991. *Information Systems Methodologies—A Framework for Understanding*. Wokingham, England: Addison-Wesley.

Oquendo, F. 1995. "SCALE: Process Modelling Formalism and Environment Framework for Goal-Directed Cooperative Processes." In *Proceedings of the IEEE International Symposium on Software Engineering Environments*, 106–124. Los Alamitos, CA: IEEE Computer Society.

Pohl, K. 1996. *Process-Centered Requirements Engineering*. New York: Wiley Research Science.

Pohl, K., K. Weidenhaupt, R. Dömges, P. Haumer, R. Klamma, and M. Jarke. 1999. "Process-Integrated Modelling Environment (PRIME): Foundations and Implementation Framework." *ACM Transactions on Software Engineering and Management* 8, no. 4: 343–410.

Pratt, M. 2001. "Introduction to ISO 10303—The STEP Standard for Product Data Exchange." *Journal of Computing and Information Science in Engineering* 1, no. 1: 102–103.

Prinz, W. 1999. "NESSIE: An Awareness Environment for Cooperative Settings." In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, ed. S. Bødker, M. Kyng, and K. Schmidt. 391–410. Dordrecht, Netherlands: Kluwer.

Ramesh, B., and V. Dhar. 1992. "Supporting Systems Development by Capturing Deliberations During Requirements Engineering." IEEE Transactions on Software Engineering 18, no. 6: 498–510.

Ramesh, B., and M. Jarke. 2001. "Towards Reference Models for Requirements Traceability." *IEEE Transactions on Software Engineering* 27, no. 1: 58–93.

Riecken, D., ed. 2000. "Personalization." Special issue, *Communications of the ACM* 43, no. 8.

Rolland, C. 1998. "A Comprehensive View of Process Engineering." In *Proceedings of the Tenth Conference on Advanced Information Systems Engineering (CAiSE'98)* (Lecture Notes in Computer Science 1413), ed. B. Pernici and C. Thanos, 1–24. Berlin: Springer.

Rosemann, M., and P. Green. 2002. "Developing a Meta Model for the Bunge-Wand-Weber Ontological Constructs." *Information Systems* 27, no. 2: 75–92.

Rossi, M. 1998. "Advanced Computer Support for Method Engineering—Implementation of CAME Environment in MetaEdit+." Ph.D. diss., Department of Computer Science and Information Systems, University of Jyväskylä.

Rossi, M., B. Ramesh, K. Lyytinen, and J.-P. Tolvanen. 2004. "Method Rationale in Method Engineering." *Journal of the Association for Information Systems* 5, no. 9: 356–391.

Rumbaugh, J., I. Jacobson, and G. Booch. 1999. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.

Russo, N., J. Wynekoop, and D. Waltz. 1995. "The Use and Adaptation of System Development Methodologies." In *Proceedings of the Sixth International Information Resources Management Association International Conference (IRMA'95)*, ed. M. Khosrowpour, 162. Hershey, PA: Idea Group.

Scheer, A.-W. 1994. "ARIS Toolset: A Software Product Is Born." *Information Systems* 19, no. 8: 607–624.

Scheer, A.-W. 1998. *ARIS—Modellierungsmethoden, Metamodelle, Anwendungen*. 3rd ed. Berlin: Springer.

Smolander, K. 1992. "OPRR—A Model for Methodology Modeling." In *Next Generation of CASE Tools* (Studies in Computer and Communication Systems), ed. K. Lyytinen and V.-P. Tahvanainen, 224–239. Amsterdam: IOS Press.

Smolander, K., K. Lyytinen, V.-P. Tahvanainen, and P. Marttiin. 1991. "MetaEdit—A Flexible Graphical Environment for Methodology Modelling." In *Advanced Information Systems Engineering* (Lecture Notes in Computer Science 498), ed. R. Andersen, J. Bubenko, and A. Sølvberg, 168–193. Berlin: Springer.

Sommerville, I., G. Kotonya, S. Viller, and P. Sawyer. 1995. "Process Viewpoints." In *Software Process Technology* (Lecture Notes in Computer Science 913), ed. W. Schäfer, 2–8. London: Springer-Verlag.

Sorenson, P. G., J.-P. Tremblay, and A. J. McAllister. 1988. "The Metaview System for Many Specification Environments." *IEEE Software* 30(March): 30–38.

Staab, S., H.-P. Schnurr, R. Studer, and Y. Sure. 2001. "Knowledge Processes and Ontologies." *IEEE Intelligent Systems* 16, no. 1: 26–34.

Sullivan, C. H. 1985. "Systems Planning in the Information Age." *Sloan Business Review* 26, no. 2: 3–11.

Teichroew, D., and E. A. Hershey III. 1977. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering* 27, no. 3: 41–48.

ter Hofstede, A. H. M., and T. P. van der Weide. 1993. "Expressiveness in Data Modeling." *Data & Knowledge Engineering* 10: 65–100.

Theodorakis, M., A. Analyti, P. Constantopoulos, and N. Spyrator. 2002. "A Theory of Contexts in Information Bases." *Information Systems* 27, no. 3: 151–192.

Tolvanen, J.-P. 1998. "Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence." Ph.D. diss., Department of Computer Science and Information Systems, University of Jyväskylä, and Jyväskylä Studies in Computer Science, Statistics and Economics.

Tolvanen, J.-P., and K. Lyytinen. 1993. "Flexible Method Adaptation in CASE—The Meta Modeling Approach." *Scandinavian Journal of Information Systems* 5, no. 1: 51–77.

Vlasblom, G., D. Rijsenbrij, and M. Glastra. 1995. "Flexibilization of the Methodology of System Development." *Information and Software Technology* 37, no. 11: 595–607.

Wand, Y. 1996. "Ontology as a Foundation for Meta-Modelling and Method Engineering." *Information and Software Technology* 38: 281–287.

Wand, Y., and R. Weber. 1989. "A Model of Systems Decomposition." In *Proceedings of the Tenth International Conference on Information Systems*, ed. J. I. DeGross, J. C. Henderson, and B. R. Konsynski, 41–51. New York: ACM Press.

Wand, Y. and R. Weber. 1993. "On the Ontological Expressiveness of Information Systems Analysis and Design Grammars." *Information Systems Journal* 3, no. 4: 217–237.

Wand, Y. and R. Weber. 1995. "On the Deep Structure of Information Systems." *Information Systems Journal* 5, no. 3: 203–223.

Westerberg, A. W. 1996. "Distributed and Collaborative Computer-Aided Environments in Process-Engineering Design." In *Proceedings of the International Conference on Intelligent Systems in Process Engineering* (American Institute of Chemical Engineers Symposium 92, no. 312), ed. J. Davis, G. Stephanopoulos, V. Venkatasubramanian, and B. Carnahan, 184–194. New York: American Institute of Chemical Engineers.

Weyhrauch, R. W. 1980. "Prolegomena to a Theory of Mechanized Formal Reasoning." *Artificial Intelligence* 13, no. 1: 133–170.

Wiederhold, G., and M. Genesereth. 1995. "The Basis for Mediation." In *Proceedings of the Third International Conference on Cooperative Information Systems (CoopIS'95)*, ed. S. Laufmann, S. Spaccapietra, and T. Yokoi, 140–155.

Winograd, T., and F. Flores. 1986. *Understanding Computers and Cognition*. Norwood, NJ: Ablex.

Yu, E. S. K. 1995. "Models for Supporting the Redesign of Organizational Work." In *Proceedings of the Conference on Organizational Computing Systems (COCS'95)*, ed. N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, 226–236. New York: ACM Press.

Yu, E. S. K., and J. Mylopoulos. 1994. "Understanding *Why* in Software Process Modeling, Analysis, and Design." In *Proceedings of the Sixteenth International Conference on Software Engineering*, Sorrento, Italy, 159–168.

Zelkowitz, M., ed. 1993. *Reference Model for Frameworks of Software Engineering Environments*. Special publication no. 500–211, National Institute of Standards and Technology, Gaithersberg, MD, and Technical report no. TR/55, Ecma International, Geneva.

Zhang, A., and K. Lyytinen. 2001. "A Framework for Component Reuse in a Metamodelling Based Software Development." *Requirements Engineering Journal* 6, no. 2: 116–131.

3 Metamodeling and Method Engineering with ConceptBase

Manfred A. Jeusfeld

slides

This chapter provides a practical guide on how to use the metadata repository ConceptBase to design information modeling methods by using metamodeling. After motivating the abstraction principles behind metamodeling, the language Telos as realized in ConceptBase is presented. First, a standard factual representation of statements at any IRDS abstraction level is defined. Next, the foundation of Telos as a logical theory is elaborated yielding simple fixpoint semantics. The principles for object naming, instantiation, attribution, and specialization are reflected by thirty-one logical axioms. After the presentation of the language axiomatization, user-defined rules, constraints and queries are introduced. The presentation of the language concludes with a description of active rules that allow the specification of reactions of ConceptBase to external events. The remainder of the chapter applies the language features described earlier in the chapter to a full-fledged information-modeling method: The Yourdan method for Modern Structured Analysis. The notations of the Yourdan method are designed according to the IRDS framework. Intranotational and internotational constraints are mapped to queries. The development life cycle is encoded as a software process model closely related to the modeling notations. Finally, aspects managing the modeling activities are addressed by metric definitions.

3.1 Introduction

The engineering of a modeling method is per se a software development effort. The result is software implementing a consistent set of modeling tools that are used to represent information about some artifacts. Modeling tools obey some underlying principles that make a dedicated environment for method engineering useful. The strongest underlying principle is the common artifact focus: The result of applying a method is a set of models all of which make statements about the same set of artifacts. Consequently, the models are interrelated. A statement in one model has implications for other statements in the same or other models about the common set of

artifacts. The artifact focus of method application regards single models as viewpoints on the artifact. A model does not contain all information about the artifact, only the information that is relevant for a particular viewpoint.

A second underlying principle is modeling life cycle. Any model has a purpose. The content of the model is the result of some modeling step and prerequisite for some other modeling, development, or analysis step. The model content must be represented in a way that is useful for the purpose.

This chapter presents a logical approach to method engineering. It uses a simple yet powerful logic to cover a number of aspects of method engineering. First, the set of allowed symbols and the allowed combination of these symbols is represented in the so-called notation level. Second, the semantics of modeling notations is investigated by incorporating a model and data level. Third, a method is defined as a combination of several notations interrelated by internotational constraints. Fourth, method application is represented by process models that prescribe or describe the modeling steps of human experts. Finally, the issue of model quality is discussed. The construction of new methods is facilitated by a multiple-perspective approach in which a high-level metamodel (the notation definition level) encodes which modeling viewpoints will be supported and which interrelationships among viewpoints require method support. The chapter uses the ConceptBase metadatabase system as method engineering tool. ConceptBase implements the logic underlying our approach and provides the necessary functions for notation definition. Furthermore, it can be used as a prototyping environment for method application. The examples in this chapter are tested with ConceptBase and are also available on the companion CD-ROM to the volume.

The chapter is organized as follows. First, the metamodeling approach is introduced, defining the four abstraction levels mentioned previously in a propositional logic. Then, the propositional statements are aggregated into framelike objects using the Telos language. After a short primer on deductive databases, we discuss predefined propositions for classification, specialization, and attribution and define their semantics through first-order logic axioms. Then, user-defined deductive rules and integrity constraints are introduced into the language framework. The description of the framework is completed with presentations of the Telos query language and of active rules. The remainder of the chapter presents a case study on engineering the Yourdan system-modeling method. It starts with engineering the entity-relationship diagramming technique in Telos using the four abstraction levels. The same principle is applied to the second major Yourdan technique, data flow diagrams. We discuss how interrelationships among notations can be represented as constraints in a query language. Some of the interrelationships among notations can be derived from common patterns of instantiation. After the definition of two notations are discussed, a software process layer is introduced that treats models as products and their creation

as development steps. The software process model itself is created from the same abstraction principles as the Yourdan notations.

3.2 Modeling Is Knowledge Representation

ConceptBase has been developed within an academic context but has been applied in both research and industrial projects. Originally, it was designed as a repository system aiding the development of data-intensive software. The modeling notations covered the whole range from requirements analysis to implementation. Hence, the repository had to be able to manage quite heterogeneous modeling languages and their interrelationships. It was decided to use the features of the Telos (Mylopoulos et al. 1990) knowledge representation language, which has its roots in requirements analysis (Greenspan 1984). As a knowledge representation language, Telos does not have a rich set of predefined features. All information in Telos has the basic format:

statement number: subject is-related-to object

A Telos statement relates two things: ''subject'' and ''object.'' The statement itself is identified by a statement number and can occur as subject or object in other statements. The simplicity of this representation is the key to its extensibility. Since the statement is identified (or *reified*), one can express statements about the statement itself. Indeed, the tokens ''subject'' and ''object'' stand for other statements defining them. The statement structure in Telos is so universal that it can represent objects, classes, metaclasses, attributes, specializations, classifications, queries, rules, and constraints.

When statements are used for modeling and metamodeling, they are assigned to IRDS levels (see section 3.12) that express their concreteness or abstractness. The following motivating example illustrates the abstraction levels:

statement 1: Bill earns 10000 dollars. (Token level)

statement 2: Employees earn salaries. (Class level)

statement 3: Entities can have attributes taken from a domain. (Metaclass level)

statement 4: Nodes are connected to nodes. (Meta-metaclass level)

Statements become more abstract from one level to the next. More specifically, each statement can be regarded as an example (= instance) of the subsequent one. This principle is called *class abstraction*. From the viewpoint of the more abstract statement, it is called *instantiation*. Rather than defining what a class is, we define the instance-to-class statement. This view allows objects to be treated completely uniformly independent of their abstraction level:

statement 5: Bill is an instance of Employee. (Token to class)

statement 6: 10000 dollars is an instance of salary. (Token to class)

statement 7: The concept Employee is an instance of the concept entity. (Class to metaclass)

statement 9: An entity is an instance of the concept node. (Metaclass to meta-metaclass)

Metamodeling as well as modeling is seen as expressing statements about an artifact. The artifact of metamodeling is a modeling environment: The metamodel expresses what provisions the modeling environment will have. Hence, metamodeling is just like any conceptual modeling of information systems. The only thing special about it is that the artifact of metamodeling is modeling. In other words, a flexible modeling environment should also be able to model modeling environments.

The Telos statement structure is so generic that it allows the expression of relations not only between "simple" objects like "Bill" and "10000" but also between statements themselves:

statement 10: statement 1 is an instance of statement 2.

statement 11: statement 1 is an instance of an attribute.

statement 12: statement 2 is an instance of an attribute.

statement 13: statement 5 is an instance of instantiation (InstanceOf).

statement 14: Bill is an ordinary object (Individual).

The last four statements are referring to predefined statements of the Telos language to express objects, their instantiation, and their attributes. There is one additional predefined statement standing for specialization (IsA). Its use is demonstrated later in the chapter.

A Telos database is essentially a semantic network in which concepts (nodes) and their interconnections (links) are treated uniformly as objects. An object can be at any level of abstraction, as motivated previously. Even abstraction levels beyond the meta-metaclass level are allowed. The feature that allows this flexibility is the explicit representation of the instance-to-class relationship. In programming languages and databases, the type of a variable or field (e.g., `INTEGER`) is visible only in the program code or schema definition. At run time, the data are held at a location in memory of suitable size. All references to the type label `INTEGER` either are compiled into memory layout or have been employed for type checking during compile time (or database creation time). In Telos, instance-to-class relationships are data them-

selves. We express them as a binary relationship[1] `(x in c)` with the reserved label `in`. The following facts encode the class abstractions of the motivating example:

```
(Bill in Employee), (10000 in Integer)
(Employee in Entity), (Integer in Domain)
(Entity in Node), (Domain in Node)
```

The foregoing example has four abstraction levels. Hence, there are three instance-to-class gaps between the objects. The instance-to-class relationship is not sufficient to express all phenomena of knowledge representation. In particular, there are relationships between objects of the same (or different) abstraction levels that are not interpreted as instantiations. For example, the objects `Bill` and `10000` are connected by a binary relationship `earns`. Obviously, this is considered to be an example of the fact that employees have salaries. In Telos, we use a predicate `(x m/l y)` to express such *attribution relationships*. We say that the object `x` is having an attribute relationship, with label `l` and category `m`, to the object `y`. The example is completed as follows:

```
(Bill salary/earns 10000)
(Employee feature/salary Integer)
(EntityType connectedTo/feature DomainOrObjectType)[2]
(Node attribute/connectedTo Node)
```

The pairing of an attribute category and an attribute label crosses two abstraction levels. For example, the category `salary` in the fact `(Bill salary/earns 10000)` is defined at the class level in the fact `(Employee feature/salary Integer)`. The expressive power of this type of pairing becomes apparent when two attributes use the same attribute category; for example:

```
(Employee feature/colleague Employee)
```

The `colleague` attribute is a feature of `Employee` just like the `salary` attribute. This phenomenon is called *multiple instantiation*: The attribute category `feature` defined as `EntityType` is instantiated multiple times for defining `Employee`. Multiple instantiation is useful at all abstraction levels; for example:

```
(Bill colleague/col1 Mary)
(Bill colleague/col2 Jim)
```

Multiple instantiation also occurs for ordinary objects as well. For example, the class `Employee` can have multiple instances:

```
(Bill in Employee)
(Mary in Employee)
(Jim in Employee)
```

The reverse application of this principle is called *multiple classification*: The same object can be an instance of multiple classes. For example, the object `Bill` may be an instance of `Employee` and also of `Pilot`:

```
(Bill in Employee)
(Bill in Pilot)
```

Besides class abstraction and attribution, the third and last structural relationship in Telos is *specialization* (the reverse: *generalization*). Whereas instantiation can be roughly compared to the element-versus-set relationship in set theory, specialization is the counterpart of the subset relationship. A specialization is denoted by a predicate `(c isA d)`. We say that `c` is defined as a *subclass* of `d` (or that `d` is a *superclass* of `c`). Specialization can be applied to objects of any abstraction level. For example, one can define a subclass `Manager` of `Employee` and a superclass `ObjectType` of `EntityType`:

```
(Manager isA Employee)
(Manager feature/salary HighInteger)
(HighInteger isA Integer)
(HighInteger in Domain)
(EntityType isA ObjectType)
```

The subclass `Manager` refines the `salary` attribute of `Employee` to a subclass of `Integer`. An instance of `Manager` is then automatically regarded as an instance of `Employee` as well:

```
(John in Manager)
(John salary/gets 500000)
(500000 in HighInteger)
```

The logical foundation of Telos as presented in section 3.6 makes sure that `(John in Employee)` as well as `(500000 in Integer)` do not need to be stated explicitly but are derivable via built-in rules of Telos. Sometimes, we include derived facts in a list of explicit facts. The semantics of the logical framework removes such duplications automatically. Indeed, the facts as shown previously are based on even more basic facts using the P-predicate (explained in section 3.6), from which they are derived.

3.3 Universal References to Objects

The factual representation of Telos statements presented in the previous section always referred to two objects and their relation. For example, the statement `(John`

in Manager) is a binary relationship between the objects John and Manager. The relationship in is interpreted as "is an instance of." Although John and Manager are object names in their own right, it is unclear whether the binary relationship between the two is also an object, that is, a statement that we can refer to. In fact, Telos allows all relationships between objects to be regarded as full-fledged objects that are allowed to participate in further relationships. For example, the object standing for (John in Manager) is an instance of another object standing for the fact that objects can be instances of other objects.[3] The problem now is that we have, as yet, no expression for the objects that are establishing a relationship (or link). This problem is overcome by the following naming convention.

Objects in Telos are either node objects or link objects. All abstraction principles then also apply to link objects. To enable this uniform application of the abstraction principles, one has to define syntax for referring to link objects, which is accomplished through the following recursive definition:

R1. The reference of a node object is its label (e.g. Employee, Bill).

R2. If an object O has the reference N and has an explicit attribute with label a, then the reference of this attribute is (N!a). The parentheses can be omitted when there is no ambiguity in the reading of the expression. Examples: Employee!salary, Bill!earns, EntityType!feature.

R3. If there are two objects O1 and O2 with references N1 and N2 and O1 is an explicit instance of O2, then (N1->N2) is the reference of the instantiation object. Examples: (Bill->Employee), (Employee->EntityType), ((Bill!earns)->(Employee!salary)).

R4. If there are two objects O1 and O2 with references N1 and N2 and O1 is an explicit subclass of O2, then (N1=>N2) is the reference of the specialization object. Example: (Employee=>Person)

The term "explicit" requires some explanation. As will be shown later, logical rules can be used to define how to derive attribute, instantiation, and specialization statements. We call a statement like (Bill in Employee) explicit if it is explicitly stored in ConceptBase. An explicit statement is regarded as an object in its own right that can have a reference (here (Bill->Employee)). Derived facts do not have this object identity property. Hence, if (Bill in Employee) is a consequence of some derivation rule, then one cannot refer to this fact: It can only occur in other rules, queries, and constraints. In particular one cannot attach further attributes to a derived fact.

With the reference conventions, just described, one can express instantiation and specialization relationships between attributes:

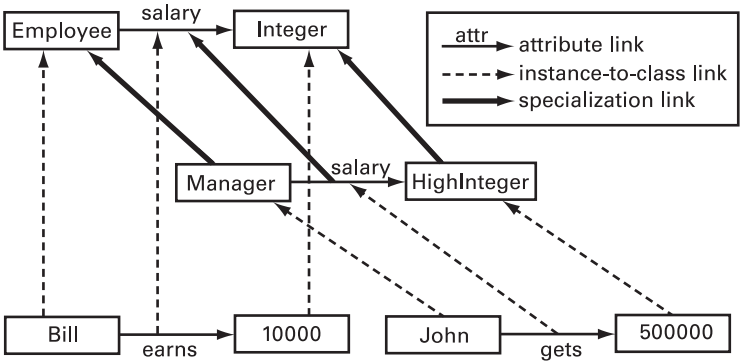


Figure 3.1
Graphical representation of Telos objects

```
(Bill!earns in Employee!salary)
(Manager feature/salary HighInteger)
((Manager!salary) isA (Employee!salary))
```

The first fact corresponds to statement 10 in the motivation. The specialization definition in the third fact can be referenced by means of a rather complicated expression. Note that a reference to an object is different from its definition. The object referred to by `((Manager!salary)=>(Employee!salary))` is indeed a specialization object between two attribute objects. On the other hand, `((Manager!salary) isA (Employee!salary))` represents the statement that there is a specialization relationship between two attributes (regardless of whether this is an explicit fact or derived by means of some rule). We use the operator # to dereference object references: If `N` is the object reference of an object `O`, then `#N` returns `O`.

Figure 3.1 shows some of the objects defined so far. Node objects appear as nodes. Instance-to-class relationships appear as broken directed links (between the instance and the class). Specialization links are thick links from the subclass to the superclass. Attributes are shown as labeled links from the object to the attribute value (which itself is an object). Note that the attribute `Manager!salary` is a subclass of the attribute `Employee!salary`. Hence, the attribute `John!gets` is also regarded as an instance of `Employee!salary` via deduction.

Our motivating example has as its highest abstraction level a meta-metaclass `Node`. It uses an attribute category `attribute` to define a link `connectedTo`. This category has not yet been defined. Indeed, we assume that a built-in object `Proposition` with an attribute `attribute` exists. It is defined as follows:

```
(Proposition in Proposition)
(Proposition attribute/attribute Proposition)
```

gel

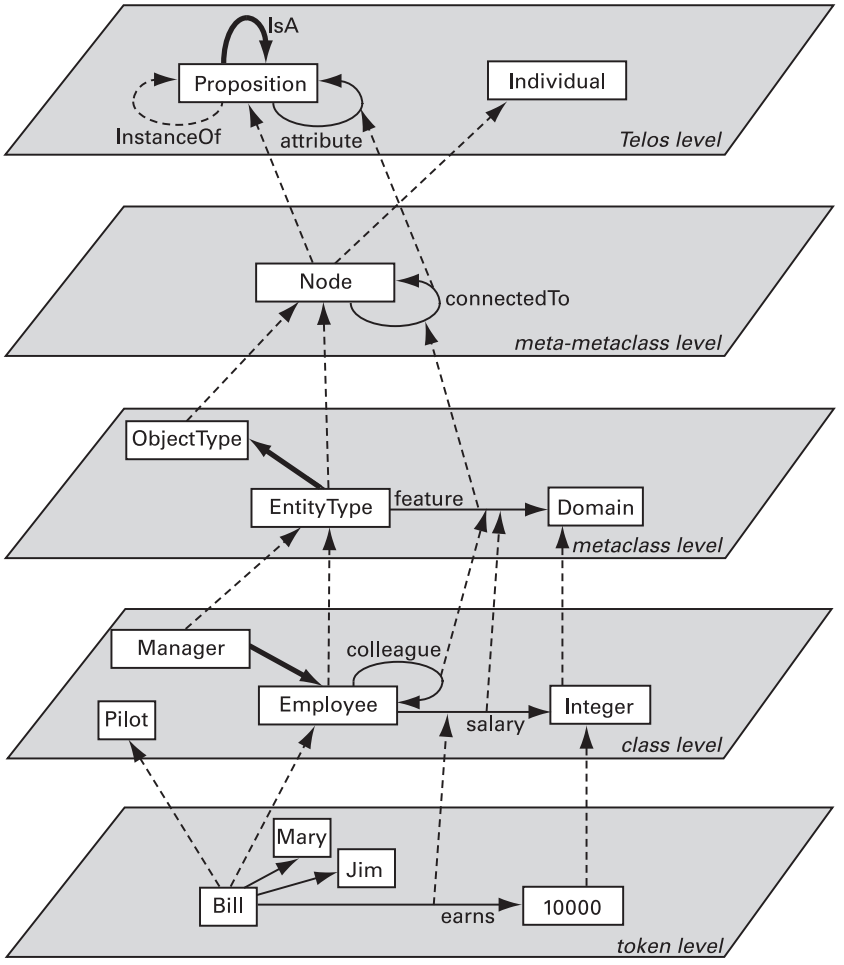


Figure 3.2
Abstraction levels in Telos

The first of these two statements is indeed a logical consequence of the built-in axioms of Telos, as shown subsequently. It makes no difference, however, if we include the statement as factual data as well. All explicit objects are automatically regarded as instances of `Proposition`, another built-in axiom of Telos. Hence, any Telos object is an instance of `Proposition` and can instantiate the attribute `attribute`.

Figure 3.2 presents the motivating example in a graphical way. It should be observed that not all definitions in the running example are shown in the figure. For example, the instantiation of `Mary` and `Jim` into `Employee` is omitted. The membership of an object in a certain IRDS level is based solely on its current instantiation. `Bill`, `Mary`, `10000`, etc., are regarded as tokens because they happen to have no instances. The object `Employee` is a (simple) class because all its instances are tokens; the object `EntityType` is a metaclass because all its instances are simple classes and so on. From a logical point of view the levels to which objects and classes are assigned are not relevant at all. Instead of insisting on "correct" assignment to levels, Telos enforces correct use of instantiation. Nevertheless, the IRDS levels shown in figure 3.2 are useful for enhancing the understandability of models that are placed at different IRDS abstraction levels.

There can well be relations other than instantiation between objects placed at different IRDS levels. Consider, for example, `(EntityType attribute/author PeterChen)`, with the intended meaning "The Employee concept was authored by PeterChen." Here, the object `PeterChen` is naturally placed at the token level, whereas the object `EntityType` is placed at the metaclass level. It is important to note that `EntityType` is just an object like any other object. It becomes a *metaclass* here only because it has instances that themselves have instances.

The definition order should also be remarked upon. ConceptBase places no restriction on the order in which objects are defined, that is, created in the metadata repository. With the *instantiation strategy*, one starts top-down at the level of meta-metaclasses (called the notation definition level in IRDS terminology), then defines metaclasses (establishing the constructs of modeling notation), and then continues with simple classes (example models) and tokens (example objects conforming to the models). In the *classification strategy*, one starts with token objects, then establishes classes to classify the tokens, then metaclasses to classify the classes, and so on. Mixed strategies are also possible and in fact are the most likely scenario, since some classes emerge only when one finds an example instance that cannot be classified into the existing set of classes. The only restriction imposed on object definition strategies is the Telos axioms stated in section 3.8. In particular, one can only create links between objects that have already been defined.

The topmost level in figure 3.2 is reserved for five predefined Telos objects. `Proposition` is the most general object and has any other object as instances, including itself. `Individual` has as instances all objects that are displayed as nodes (i.e. not

as attributes, instantiations, or attributions). The `InstanceOf` link of `Proposition` has all explicit instantiations as instances; for example `Bill->Employee` and `(Employee!salary->EntityType!feature)`. The `IsA` link has all specializations as instances; for example `Manager=>Employee`. Finally, the `attribute` link has all attributes as instances. The precise definitions of these five objects are presented in section 3.8.

3.4 The Telos Frame Syntax

The definitions in the preceding sections employ the predicate notation that is part of the query language of ConceptBase. The advantage of such a logical representation is its preciseness. However, readability suffers in this type of representation, as all information is decomposed into small pieces. *Frame* syntax is an alternate means of expression that eliminates this problem. It provides a denser representation of object definitions by grouping all information regarding one object into one textual frame. In this section, we use the motivating example to introduce frame syntax:

```
Bill in Employee,Pilot with
  salary
    earns: 10000
  colleague
    col1: Mary;
    col2: Jim
end

Mary in Employee end
Jim in Employee end

John in Manager with
  salary
    gets: 500000
end

500000 in HighInteger end
```

In the example, the attribute category `salary` precedes the definitions of the attribute. If more than one attribute falls under the same category, then a semicolon separates them. A comma separates multiple classes of an object. Note that objects like `Mary` can exist without instantiating all or even any attribute category of their class.

The frame definition of `Employee` shows that the same syntax is applied for class definitions:

source

```
Employee in EntityType with
  feature
    salary: Integer;
    colleague: Employee
end
Manager in EntityType isA Employee with
  feature
    salary: HighInteger
end

Pilot in EntityType end
Integer in Domain end
HighInteger in Domain isA Integer end
Employee in Domain end
```

Superclasses are declared within the definition of the subclass of which they are a superclass. The subclass `Manager` in the example refines the attribute `salary` of `Employee`. In such cases the attribute value `(HighInteger)` at the subclass level must be a subclass of the corresponding attribute value `Integer` of the superclass `Employee`. If a class has more than one superclass, then commas separate them.

We include a metaclass `DomainOrObject` as a superclass of `Domain` and `ObjectType`:

```
EntityType in Node isA ObjectType with
  connectedTo
    feature: DomainOrObjectType
end
ObjectType in Node isA DomainOrObjectType end
Domain in Node isA DomainOrObjectType end
DomainOrObjectType in Node end
```

Because of the inclusion of this metaclass, an instance of `EntityType` can have either a domain as feature (e.g., the attribute `salary`) or a reference to an object type (e.g., the `colleague` attribute). The example is completed by the definition of the meta-metaclass:

```
Node in Proposition with
  attribute
    connectedTo: Node
end
```

Figure 3.3 displays the example using the ConceptBase graph browser. The labeled thin links are instantiations, the thick links are specializations, and the unlabeled thin links are ordinary attributes.

gel

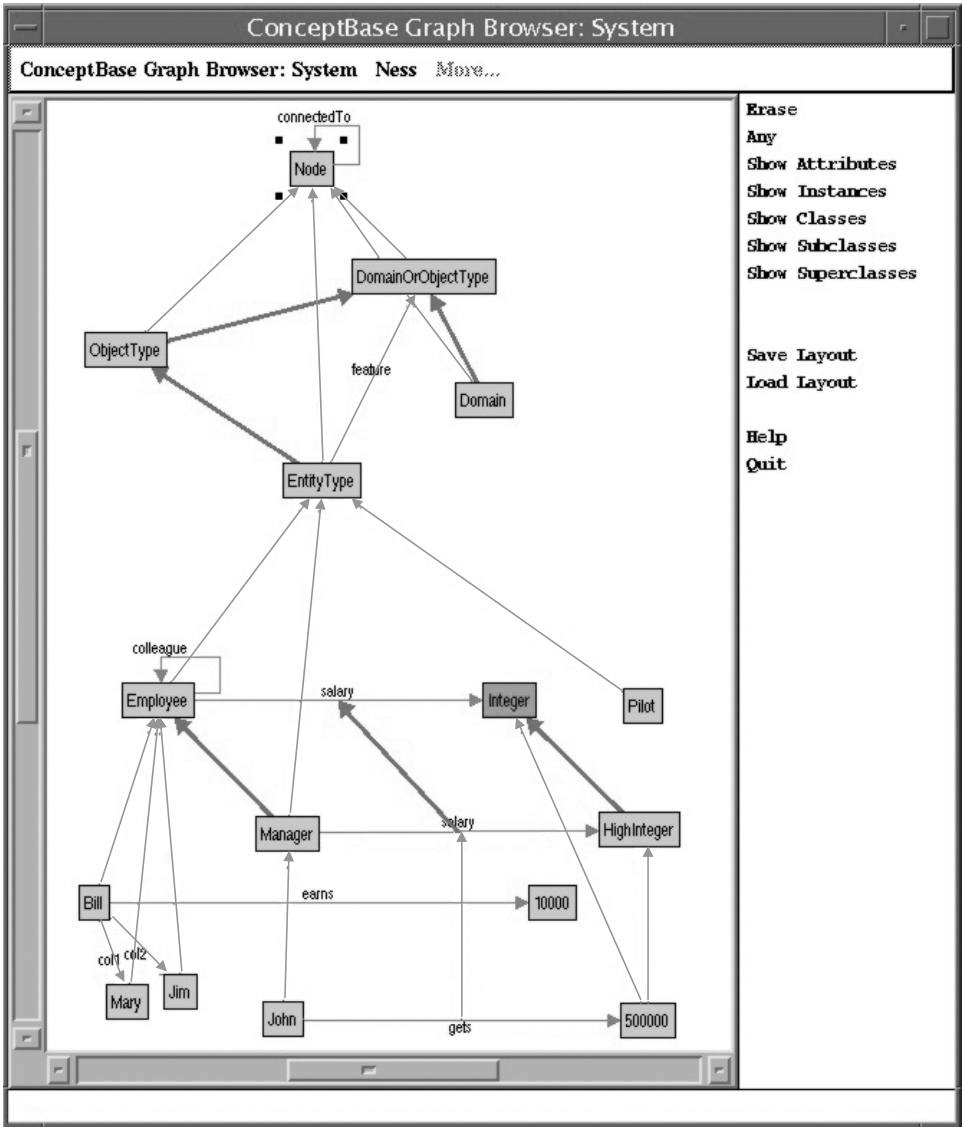


Figure 3.3
Display of the Telos example in the ConceptBase graph browser

3.5 A Short Primer in Logic for Databases

slides1

slides2

The implementation of Telos in ConceptBase is based on a simple logic called Datalog. The advantage of relying on Datalog is that the semantics of a Telos model can be precisely defined. It is also the foundation of the expressive query language in ConceptBase and its extensibility at all abstraction levels. A few basic concepts of Datalog are essential to fully understand the Telos query and rule language.

I assume here that the reader has some initial knowledge of the relational-database model and of predicate logic. In Datalog, a database is defined as a pair `(DB,R)`, where `DB` is the set of *base relations* and `R` is a set of *deductive rules*. A base relation is referred to in logic by a predicate `R(x1,x2,...,xk)`, as specified in the domain calculus of relational-database theory. A deductive rule has the form

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> S(x1,x2,...)
```

Unlike in standard Datalog notation, the logical quantifier `forall` is made explicit here.[4] The predicates R_i are positive literals of the condition; the predicates Q_i are negative literals. The number `m` may be zero; that is, a deductive rule without a negative literal is allowed. There is exactly one conclusion literal, the predicate `S`. All variables are universally quantified. If a variable occurs in a negative literal of the condition, it must also occur in a positive literal of the condition. A deductive rule in which the conclusion predicate also occurs in the condition is referred to as a *recursive rule*.

The interpretation of a deductive database is based on Herbrand semantics. In this type of semantics, a constant like `10` is interpreted by itself.[5] A *ground* (= variable-free) occurrence of a predicate is also interpreted by itself. A rule with empty conditions must be a ground. Such a degenerated rule is also called a *fact*. We assume that all facts are in `DB`, that is, that `R` does not contain rules with empty conditions. Datalog allows only constants or variables as arguments of predicates. Then, a deductive database `(DB,R)` is interpreted as the smallest set of facts (the *fixpoint*) fulfilling the following conditions:

1. If `R(c1,...,ck)` is a fact in a base relation of `DB`, then `R(c1,...,ck)` is in the fixpoint.
2. If there is a combination `c1`, `c2`, ... of constants and a rule `R`

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> S(x1,x2,...)
```

such that the substitution `s=[x1=c1,x2=c2,...]` yields facts `Ri[s]` that are already in the fixpoint and `Qj[s]` that are not in the fixpoint, then `S(c1,c2,...)` is also in the fixpoint.

A *substitution* is an operator that replaces variables in a formula with terms (in this case, with constants). Computation of the fixpoint requires that recursive rules fulfill a *stratification* condition. Stratification refers to a particular assignment of levels (numbers) to predicates. A predicate referring to a base relation gets the level 0. A conclusion predicate must be assigned a level that is greater than or equal to the level of all positive literal occurrences in the condition and strictly greater than all negative literal occurrences in the condition. If levels are assigned in this way, then the rule set `R` is said to be stratified. One can prove that when a rule set is stratified, there is a unique smallest fixpoint, the *perfect model*.

An example of a nonstratifiable rule set consisting of just one rule is

```
forall x P(x) and not Q(x) ==> Q(x)
```

Intuitively, nonstratifiable rule sets express paradoxes.[6] Consider for this the definition of a set `M` as the set of all sets that do not contain themselves as elements. In logic, this is expressed as equivalence:

```
forall s Set(s) and not Element(s,s) <==> Element(s,M)
```

We can read this formula as the conjunction of two logical implications:

```
forall s Set(s) and not Element(s,s) ==> Element(s,M)

forall s Element(s,M) ==> Set(s) and not Element(s,s)
```

The first of these implications is not stratifiable, because the implied predicate occurs negatively in the condition.

Integrity constraints in deductive databases are special rules of the form

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> inconsistent
```

A deductive database `(DB,R)` is said to be consistent if the fact `inconsistent` is not in its fixpoint, that is, if the fact `inconsistent` cannot be derived. The predicate `inconsistent` may not occur in a condition of any rule. We assume that an update to a base relation may not lead to inconsistency. If `inconsistent` is derivable after an update, the update is rejected, and the database is rolled back to the state before the update.

Integrity constraints are useful for guaranteeing that certain database states are never reached because they represent errors. In modeling and design environments, we are sometimes less strict when dealing with integrity constraints: Rather than forbidding certain database states, we are interested in minimizing the number of violations, with the goal of zero violations at the end of the modeling process. The rationale behind this viewpoint is that the database is incomplete at the start of the modeling process, and the integrity constraint violations are removed as the database containing the models becomes more and more complete. If we employed traditional integrity constraints, then the incomplete early database states would be rejected, which would be unacceptable in a modeling environment. A way out of this dilemma is to define ordinary deductive rules that derive violations that formally are not regarded as integrity violations:

```
forall x1,x2,...
R1(x11,...) and ... Rn(xn1,...) and
not Q1(y11,...) and ... not Qm(ym1,...)
==> violator(x1,x2,...)
```

Current violations in the database can be computed by querying the `violator` predicate. The solutions indicate to the modeler where the current database has to be extended or modified in order to reach fewer violations.

Example 3.1 Consider the following rule set and check it for stratifiability (The predicate `R(.,.)` refers to a base relation.)

```
forall x P(x,y) and R(x,z) and not Q(x,z) ==> L(z)
forall x,y L(x) and L(y) ==> Q(x,y)
```

The base relation `R` gets the lowest stratification level:

```
strat(R)=0
```

The first rule induces the following conditions:

```
strat(P) ≤ strat(L)
strat(Q) < strat (L)
```

The second rule adds the condition

```
strat(L) ≤ strat(Q)
```

Thus, there is no stratification for this rule set.

Example 3.2 Consider the rule set

```
forall x P(x,y) and R(x,z) and Q(x,z) ==> L(z)
forall x,y L(x) and L(y) ==> Q(x,y)
```

This rule set is stratifiable:

```
strat(R)=strat(P)=strat(L)=strat(Q)=0
```

Example 3.3 Let `move(.,.)` be a base relation. Consider the rule set consisting of a single rule:

```
forall x,y move(x,y) and not win(y) ==> win(x)
```

This rule set is obviously not stratifiable, because stratification would require `strat(win) < strat(win)`. The example is, however, interesting, as it encodes "win" positions of simple games. If the move database is acyclic, that is, if there is no cyclic path in the database matching

```
move(x1,x2),....,move(xk,x1)
```

then no fact `win(x)` is derived from the negation of itself. Although the rule is not stratifiable, we still can compute a unique fixpoint. Hence, there are cases in which the static stratification test described previously is stricter than necessary.

The companion CD-ROM to this volume contains the "win" example in Telos and shows some further interesting applications of nonstratified rule sets. More on the theoretical foundations of Datalog and its extensions can be found in Ceri, Gottlob, and Tanca 1990 and Chen and Warren 1996.

3.6 The Logical Foundation for Telos

The Telos object definitions shown up to now have been just pieces of text in a certain syntax. One could imagine that they were stored in a file and then looked up by a standard word processor. The Telos language becomes useful only if some automation is offered. In ConceptBase, this automation is primarily founded on a logic-based facility that allows analysis of large sets of Telos definitions (also called a Telos model). This facility comes in three flavors:

1. A *constraint* expresses a necessary condition that has to be fulfilled by the Telos model. For example, one might specify that no employee may earn more than her boss.
2. A *deductive rule* can be used to derive new predicate facts from existing ones. For example, the boss of an employee may be derived from the head of the department where the employee works.
3. A *query class* is syntactically a Telos class with a constraint definition. However, the constraint is regarded not as a necessary condition, but as a sufficient condition, for an object's being an instance of the class.

We introduced Telos through three predicates: `(x in c)` for class abstraction, `(c isA d)` for generalization, and `(x m/l y)` for attribution. The key to Telos, however, is that all these predicates are derived from a single base predicate, the *P-predicate* or "proposition":

```
P(o,x,l,y)
```

The P-predicate is used to represent all explicit Telos objects. The component `o` is called the object identifier, `x` is the source, `l` the label, and `y` the destination of the object. The P-predicate is used to define the predicates mentioned in the foregoing by means of four axioms:

```
forall o,x,c P(o,x,in,c) ==> (x in c)
forall o,c,d P(o,c,isa,d) ==> (c isA d)
forall o,x,l,y,p,c,m,d P(o,x,l,y) and P(p,c,m,d) and (o in p) ==>
(x m/l y)
forall x,m,l,y (x m/l y) ==> (x m y)
```

The axioms should be read as deductive rules: If the condition before the implication sign is true for some substitution of variables, then the correspondingly substituted predicate after the implication sign is true. We assume that these four axioms are predefined, that is, that they are axioms of the logical theory (being a set of logical formulas). We define a Telos database as a triple `(OB,IC,R)` where

- `OB` $\subseteq$ `{P(o,x,l,y)|o,x,y object identifier,l label}` is the finite extensional database of Telos objects;
- `IC` is a finite set of integrity constraints;
- `R` is a finite set of deductive rules.

An element of `OB` is called a *P-fact* or Telos object. We refer to an occurrence of `P(o,x,l,y)` as a P-predicate. We then assume that it occurs inside a logical formula in which some parameters of the P-predicate can be logical variables.

The axioms discussed in this chapter are predefined entries in `IC` or `R`. Any object in a Telos database must be an instance of some class. It has already been noted that the object with name `Proposition` is predefined in Telos (in `OB`). Indeed, at least the following five objects must be predefined in order to be able to define the semantics of the Telos abstraction principles instantiation, attribution, and specialization:

```
P(p1,p1,Proposition,p1)
P(p2,p2,Individual,p2)
P(p3,p1,attribute,p1)
P(p4,p1,InstanceOf,p1)
P(p5,p1,IsA,p1)
```

For reasons of readability, we occasionally use expressions like `#Individual` to refer to the identifier of an object like `Individual`. The expression stands for the object identifier of `Individual`, here, `p2`. The operator # is omitted when it is clear

that we refer to the object with the given name. The purpose of the five predefined objects becomes apparent with the following axioms (to be read as deductive rules):

```
forall o,x,l,y P(o,x,l,y) ==> (o in #Proposition)
forall o,l P(o,o,l,o) ==> (o in #Individual)
forall o,x,c P(o,x,in,c) ==> (o in #InstanceOf)
forall o,c,d P(o,c,isa,d) ==> (o in #IsA)
forall o,x,l,y P(o,x,l,y) and (o \== x) and (l \== in) and (l \==
isa) ==> (o in #Proposition!attribute)
```

These five axioms ensure that we automatically know that each object is an instance of the class `Proposition` and that the membership of the four other classes is based on the structure of the P-predicate. Where possible, we use the reference of an object (e.g., `Proposition`) instead of the object identifier `(p1)` to enhance readability. Object references are precisely defined in section 3.3.

The predicate `(c isA d)` for subclasses is supposed to be both reflexive and transitive. Moreover, any instance of a subclass must also be an instance of any superclass of that subclass:

```
forall o (o in Proposition) ==> (o isA o)
forall c,d,e (c isA d) and (d isA e) ==> (c isA e)
forall x,o,c,d (x in c) and P(o,c,isa,d) ==> (x in d)
```

Finally, instances of classes must use the attribute definitions of the class in a conforming way (this is referred to as the attribute-typing axiom):

```
forall o,x,l,y,p,c,m,d P(o,x,l,y) and P(p,c,m,d) and (o in p) ==>
(x in c) and (y in d)
```

The preceding formula is a predefined constraint of `IC`. Consider, as an example, the attribute `Bill!earns`, which is an instance of `Employee!salary`. Then the constraint forces `Bill` to be an instance of `Employee` and `10000` (the attribute value of `Bill!earns`) to be an instance of `Integer` (the attribute value of `Employee!salary`).

There are additional built-in constraints on attribute specialization and on preventing malformed P-facts. They are omitted here because they are mainly of a technical nature. The complete list of axioms is presented in section 3.8. All axioms can be transformed into Datalog with stratified negation.

3.7 From Frames to Objects and Vice Versa

The frame syntax presented in the foregoing is the standard way to express textual definitions of Telos objects. A frame indeed clusters class membership, superclasses,

and attributes into one textual object. In order to understand the precise meaning of frames, one has to define a mapping of frames to P-predicates, because the semantics of relationships like specialization are defined in terms of P-predicates. As an example, consider the frame

```
EntityType in Node isA ObjectType with
  connectedTo
    feature: Domain
end
```

In the first step of the mapping the frame is rewritten into flat predicate facts for classification, specialization, and attribution. It should be noted that predicate facts establish binary relationships between named objects. Attribution is a special case, since it has two labels as infix symbols: first the label of the attribute itself, and second the label of the *attribute category* of the attribute.

```
(EntityType in Node)
(EntityType isA ObjectType)
(EntityType connectedTo/feature Domain)
```

In the second step, each predicate fact is considered individually, and we look for an axiom in `R` whose conclusion predicate matches the predicate fact. We consider only the following three axioms:

```
D1. forall o,x,c P(o,x,in,y) ==> (x in c)
D2. forall o,c,d P(o,c,isa,d) ==> (c isA d)
D3. forall o,x,l,y,p,c,m,d P(o,x,l,y) and P(p,c,m,d) and (o in p)
==> (x m/l y)
```

Before mapping a predicate fact to a P-fact, one has to check whether a fact like `(EntityType in Node)` can be derived (for example, via the class membership axiom of the specialization relationship). If not, we match the fact `(EntityType in Node)` with the conclusion predicate `(x in c)` of rule `D1`, leading to a variable substitution `[x=EntityType,c=Node]`. In order to make `(EntityType in Node)` deducible via rule `D1`, the predicate `P(o,#EntityType,in,#Node)` must be true for some object that can serve as a value of variable `o`. If the database `OB` already contains such an object, then nothing has to be done. If not, then a new object identifier like `o1` has to be generated and the object `P(o101,#EntityType,in,#Node)` can be inserted into `OB`. In the same way, `(EntityType isA ObjectType)` is mapped to an object `P(o102,#EntityType,isa,#ObjectType)`.

The third fact established in step 1 `(EntityType connectedTo/feature Domain)` is an attribute definition. It is matched against rule `D1`, leading to a substitution `[x=EntityType,m=connectedTo,y=Domain,l=feature]`. Its mapping to

P-predicates is more complex than that of the instantiation and specialization predicates, because the attribute-typing axiom requires that we exclude attributes that do not conform to the attribute definition at the class level. So, for the partially substituted predicate `P(p,c,connectedTo,d)`, the predicates `(EntityType in c)` and `(Domain in d)` must be derivable. Consequently, we can limit the search for `P(p,c,connectedTo,d)` accordingly. Theoretically, more that one potential result for the search may exist, since the same object `x` can be an instance of multiple classes. However, the built-in Telos axiom A17 (see section 3.8) prevents such ambiguity. Once values for the variables in `P(p,c,connectedTo,d)` have been found, we can insert a new object `P(o103,#EntityType,feature,#Domain)` into the database, with `o103` substituting for the variable `o`. Given the current state of the database, the resulting substitution is `[p=o11,c=Node,d=Node]`. The obligation to make `(o103 in o11)` true remains. Since `o11` is a constant, we can use the first rule, `D1`, to do so. Hence, the resulting Telos objects are

```
P(o101,#EntityType,in,#Node)
P(o102,#EntityType,isa,#ObjectType)
P(o103,#EntityType,feature,#Domain)
P(o104,#EntityType!feature,in,#Node!connectedTo)
```

It should be noted that for the sake of efficiency and uniformity, all object references are resolved on the fly against the object identifiers. For example, the reference `EntityType!feature` is resolved to `o103`. P-fact `o104` shows how attribute categories are treated in Telos: They are mapped to instantiation relationships between an attribute object and another attribute object. After object references are replaced with identifiers, the stored P-facts will be

```
P(o101,o12,in,o11)
P(o102,o12,isa,o13)
P(o103,o12,feature,o14)
P(o104,o103,in,o15)
```

where we assume the existence of

```
P(o11,o11,Node,o11)
P(o12,o12,EntityType,o12)
P(o13,o13,ObjectType,o13)
P(o14,o14,Domain,o14)
P(o15,o11,connectedTo,o11)
```

The mapping from Telos frames to objects can be used directly for implementing the "insert" operation, that is, adding new objects to the Telos database. A similar method can be employed when one wants to remove a Telos frame from the

database. First, the frame is mapped to P-facts, and then the P-facts are removed from the database.

Remark 3.1 Since only some axioms of Telos are here, the complete method for translating a Telos frame object is a bit more complex than the preceding discussion indicates. For example, a subclass may refine an attribute that is already defined in a superclass of the subclass, like the `Manager!salary` attribute. In such a case, an instance of `Manager` would instantiate the more specific attribute, whereas an instance of `Employee` would instantiate the more general attribute. The theoretic framework for this mapping method is called *abduction*. Abduction is an approach in which a user can specify which derived predicates can be inserted (i.e., should be derivable after an update) and which can be deleted (i.e., should no longer be derivable after the update). Abduction has to cope with the problem that for the same predicate, multiple deductive rules can exist. Hence, the mapping from frames to P-facts is in principal ambiguous, and one has to develop notions of minimality to prioritize the rule selection. In our case, we consider only the three "basic" rules for the instantiation, attribution, and specialization predicates (see formulas D1,D2,D3 earlier in this section).

It should be noted that this mapping method allows Telos frames to be inserted where certain attributes are already derivable from the database. In such cases, no new objects are inserted. Because of this, we call the method `Tell` and the reverse method `Untell`. Users of the ConceptBase system can display the P-facts generated (removed) for a `Tell` (`Untell`) operation by setting the parameter trace mode to "veryhigh." See the ConceptBase user manual (Jarke, Jeusfeld, and Quix 2003) for instructions.

The reification of all Telos statements is the precondition for using Telos for method engineering, in particular for defining the semantics of certain symbols in modeling notations. The treatments of attributes as ordinary objects allows rules and constraints to be formulated for such objects regardless of whether they are at data level, model level, notation level, or an even higher level. The list of Telos axioms presented in the next section effectively defines the semantics of the three Telos relationship types: instantiation, specialization, and attribution. User-defined relationship types and the definition of their semantics are presented in section 3.9.

3.8 Telos Predicates and Axioms

Telos has a single base relation, the P-facts, on which further predicates are defined via logical axioms. Some of these axioms are deductive rules; others are constraints that forbid the occurrence of certain combinations of P-facts. The list of Telos predicates is as follows:

`P(o,x,n,y)`

This predicate ranges directly over the P-facts. We say: there is a P-fact identified by `o` that links two P-facts identified by `x`, `y`; the link is labeled `n`.

`From(o,x)`

There is a P-fact identified by `o` whose second argument is `x` (the *source* of object `o`).

`Label(o,n)`

There is a P-fact whose third component is `n` (the *label* of object `o`).

`To(o,y)`

There is a P-fact identified by `o` whose fourth argument is `y` (the *destination* of object `o`).

`(x in c)` or `In(x,c)`

Object `x` is an instance of class `c`, or in other words, `c` is a class to which object `x` belongs.

`(c isA d)` or `Isa(x,c)`

Class `c` is a subclass of class `d`, or in other words, class `d` is a superclass of class `c`.

`(x m/n y)` or `AL(x,m,n,y)`

There are two objects `x` and `y` that are linked to each other by a P-fact that has label `n`. That P-fact is an instance of another object, which has the label `m`. We say that `m` is the *category* of the link between `x` and `y`.

`(x m y)` or `A(x,m,y)`

There are two objects `x` and `y` that are related to one another by a binary relationship `m`.

`Aid(x,m,o)`

There is a P-fact `o` that has `x` as its source and has category `m`.

`(x < y), (x > y), (x = y), (x =< y), (x >= y)`

Numerical comparison between objects `x` and `y`. Both objects must be instances of the built-in classes `Integer` or `Real`.

`(x == y)` or `IDENTICAL(x,y)` or `UNIFIES(x,y)`

The two objects `x` and `y` are the same.

Thirty-one rules and constraints are predefined in the variant of Telos presented here (Jeusfeld 1992). To distinguish this axiomatization of Telos from earlier Telos

definitions, we sometimes refer to it is *O-Telos*. The O-Telos axioms precisely define what we understand by instantiation, specialization, and attribution. Furthermore, they define the extension (= logical interpretation) of some of the predicates listed previously.

Axiom 3.1 Identifiers of P-facts are unique, or in other words, the first field of P-facts is a key for the remaining fields.

```
forall o,x1,n1,y1,x2,n2,y2
P(o,x1,n1,y1) and P(o,x2,n2,y2) ==> (x1=x2) and (n1=n2) and
(y1=y2)
```

Axiom 3.2 The name of an individual object is unique.

```
forall o1,o2,n
P(o1,o1,n,o1) and P(o2,o2,n,o2) ==> (o1=o2)
```

Axiom 3.3 Names of attributes are unique in conjunction with the source object, or in other words, no object may have two attributes with the same name. This does not necessarily hold for instantiation and specialization objects (see axiom 3.4).

```
forall o1,x,n,y1,o2,y2
P(o1,x,n,y1) and P(o2,x,n,y2) ==> (o1=o2) or (n=in) or (n=isa)
```

Axiom 3.4 The name of instantiation and specialization objects (labels `in`, `isa`) is unique in conjunction with source and destination objects.

```
forall o1,x,n,y,o2 P(o1,x,n,y) and P(o2,x,l,y) and ((n=in) or
(n=isa)) ==> (o1=o2)
```

Axiom 3.5 Instantiation objects lead to solutions for the `In` predicate.

```
forall o,x,c P(o,x,in,c) ==> In(x,c)
```

Axiom 3.6 Specialization objects induce a specialization relationship `Isa` between the two referenced objects.

```
forall o,c,d P(o,c,isa,d) ==> Isa(c,d)
```

Axiom 3.7 If there is an attribute with name `n` between two objects `x` and `y`, and this attribute is an instance of an attribute class with name `m`, then a solution for the `AL` predicate can be derived:

```
forall o,x,n,y,p,c,m,d P(o,x,n,y) and P(p,c,m,d) and In(o,p) ==>
AL(x,m,n,y)
```

Axiom 3.8 The ordinary attribute predicate `A` is based on the `AL` predicate by omitting the attribute label `n`.

```
forall x,m,n,y AL(x,m,n,y) ==> A(x,m,y)
```

Axiom 3.9 If an object uses an attribute label from the class level, then it must actually be from one of the classes of the object. If some attribute predicate can be derived for x, then it will be due to an instantiation of an attribute category defined at the level of a class to which `x` belongs. Note that solutions for the `A` predicate can be obtained only from the Telos axioms. User-defined rules do not interfere with this axiom, as their attribution predicates are formally distinguished from the `A` predicate.

```
forall x,y,p,c,m,d In(x,c) and A(x,m,y) and P(p,c,m,d) ==> exists
o,n P(o,x,n,y) and In(o,p)
```

Axiom 3.10 The `isa` relation is reflexive. The object identifier `#Proposition` (= `p1`) is declared in axiom 3.24.

```
forall c In(c,#Proposition) ==> Isa(c,c)
```

Axiom 3.11 The `isa` relation is transitive.

```
forall c,d,e Isa(c,d) and Isa(d,e) ==> Isa(c,e)
```

Axiom 3.12 The `isa` relation is antisymmetric.

```
forall c,d Isa(c,d) and Isa(d,c) ==> (c=d)
```

Axiom 3.13 Class membership of objects is inherited upwardly to superclasses. This is the only "inheritance rule" in Telos. Inheritance of attributes by subclasses is a redundant principle that is subsumed via axioms 3.13 and 3.14: Any instance of a subclass is also an instance of the superclasses of which it is a subclass (axiom 3.13) and thus can instantiate the attributes of those superclasses (axiom 3.14).

```
forall p,x,c,d In(x,d) and P(p,d,isa,c) ==> In(x,c)
```

Axiom 3.14 Instance attributes are "typed" by attributes defined at class level.

```
forall o,x,l,y,p P(o,x,l,y) and In(o,p) ==> exists c,m,d
P(p,c,m,d) and In(x,c) and In(y,d)
```

Axiom 3.15 Subclasses that define attributes with the same name as attributes of their superclasses must refine these attributes. Hence, the attribute definition of a superclass is never violated by a refinement of that attribute at subclass level. Specifically, the refined attribute is a specialization of the superclass attribute, and the attribute value of the refined attribute is a specialization of its counterpart at superclass level.

```
forall c,d,a1,a2,m,e,f Isa(d,c) and P(a1,c,m,e) and P(a2,d,m,f)
==> Isa(f,e) and Isa(a2,a1)
```

Axiom 3.16 If an attribute is a refinement (in a subclass) of another attribute, then it must also refine the source and destination components of the corresponding P-fact of the attribute. Note that the two attributes will not necessarily have the same label,

as was the case in axiom 3.15. However, all refinements of the type specified in axiom 3.15 are also governed by axiom 3.16 because of the implied truth of `Isa(a2,a1)` in axiom 3.15.

```
forall c,d,a1,a2,m1,m2,e,f
Isa(a2,a1) and P(a1,c,m1,e) and P(a2,d,m2,f) ==> Isa(d,c) and
Isa(f,e)
```

Axiom 3.17 For any object there is always a unique "smallest" attribute class with a given label `m`. Hence, whenever two different attributes with the same label are applicable to an instantiated object `x`, then there is a third attribute `a3` which is specializing both other attributes `a1`, `a2`.

```
forall x,m,y,c,d,a1,a2,e,f
In(x,c) and In(x,d) and P(a1,c,m,e) and P(a2,d,m,f) ==> exists
g,a3,h In(x,g) and P(a3,g,m,h) and Isa(g,c) and Isa(g,d)
```

Axiom 3.18 Any object is an instance of the class `Proposition`. Any instance of `Proposition` is an object.

```
forall o,x,n,y P(o,x,n,y) <==> In(o,#Proposition)
```

Axiom 3.19 Any individual object is an instance of the class `Individual`. Any instance of the class `Individual` must be an individual object. The operator `\==` stands for inequality of object identifiers. Note that an individual object is determined by its structure; that is, its identifier, source, and destination are the same.

```
forall o,l P(o,o,n,o) and not (n \== in) and not (n \== isa) <==>
In(o,#Individual)
```

Axiom 3.20 All instantiation objects are instances of the `InstanceOf` attribute of `Proposition`. Any instance of the `InstanceOf` attribute of `Proposition` must be an instantiation object.

```
forall o,x,c P(o,x,in,c) and (o \== x) and (o \== c) <==>
In(o,#Proposition!InstanceOf)
```

Axiom 3.21 All specialization objects are instances of the `InstanceOf` attribute of `Proposition`. Any instance of the `InstanceOf` attribute of `Proposition` must be a specialization object.

```
forall o,c,d P(o,c,isa,d) and (o \== c) and (o \== d) <==>
In(o,#Proposition!IsA)
```

Axiom 3.22 All attribute objects are instances of the `attribute` attribute of `Proposition`. Any instance of the `attribute` attribute of `Proposition` must be an attribute object.

```
forall o,x,n,y P(o,x,l,y) and (o \== x) and (o \== y) and (n \==
in) and (n \== isa) <==> In(o,#Proposition!attribute)
```

Axiom 3.23 Any object (i.e., any P-fact) is either an individual object, an instantiation relationship, a specialization relationship, or a regular attribute. Note that the axioms 3.22 and 3.23 exclude P-facts like `P(o,o,n,p)` and `P(o,p,n,o)`.

```
forall o In(o,#Proposition) ==>
In(o,#Individual) or In(o,#Proposition!InstanceOf) or
In(o,#Proposition!IsA) or In(o,#Proposition!attribute)
```

Axioms 3.24–3.28 There are exactly five built-in classes (the Telos built-in classes). ConceptBase has many more predefined classes to manage itself and to provide data structures for its functionality, in particular, for user-defined rules, constraints, and queries. (The object identifiers `p1` to `p5` used here are arbitrary. Any other set of five different identifiers will also work.)

```
P(p1,p1,Proposition,p1)
P(p2,p2,Individual,p2)
P(p3,p1,attribute,p1)
P(p4,p1,InstanceOf,p1)
P(p5,p1,IsA,p1)
```

Axiom 3.29 Objects must be defined before they are referenced. The operator `*prec` is some predefined total order on the set of object identifiers.

```
forall o,x,n,y P(o,x,n,y) ==> (x *prec o) and (y *prec o)
```

Axiom Schema 3.30 Let `P(p,c,m,d)` be an arbitrary object. Then any binary instantiation predicate `In(o,p)` leads to a solution for its unary version `In.p(o)`.

```
forall o In(o,p) ==> In.p(o)
```

Axiom Schema 3.31 Let `P(p,c,m,d)` be an arbitrary object. An attribute linking objects `x` and `y` that is an instance of object `p` induces a binary attribute predicate `A.p`.

```
forall o,x,l,y P(o,x,l,y) and In(o,p) ==> A.p(x,y)
```

The last two axiom schemas distinguish the `In` and `A` predicates as defined by the Telos axioms from those declared in user-defined rules and constraints. User-defined rules are always transformed into versions that use `In.c` and `A.p` predicates instead of `In` and `A` predicates. Through this transformation, the definition of `In` and `A` cannot be altered by user-defined rules.

User-defined rules are further restricted to `In.c` and `A.p` predicates in their conclusions. This completely shields the axioms from any further definition made by a

user. For the sake of readability, the user-defined formulas use predicates in their original syntax, such as

`In(x,c)` or `(x in c)`
`A(x,m,y)` or `(x m y)`

Internally, all such occurrences are replaced by predicates `In.c` or `A.p`, respectively. Thus, internally each class object and each attribute category has its own predicate symbol. Without this transformation, only a few rule sets would be stratifiable, as a result of the small number of predicate symbols.

Stratification as explained in section 3.5 is based on predicate names, that is, independent of the arguments of a predicate occurrence in a formula. This type of stratification is also called *static stratification*. *Dynamic stratification* extends the principle to predicate occurrences including their arguments. Like its static counterpart, dynamic stratification guarantees perfect-model semantics. Any statically stratified Datalog theory, i.e., any set of statically stratified Datalog rules, is also dynamically stratified, but not vice versa. For the purpose of this book, it is sufficient to assume static stratification. I would like to mention, however, that ConceptBase supports dynamic stratification, which is tested when a formula is evaluated rather than when it is defined. The CD-ROM accompanying this volume includes some examples of dynamically stratified Telos models as well as examples of Telos models that are not dynamically stratified.

Axioms 3.1 to 3.31 can be represented in Datalog either as deductive rules or integrity constraints or as a combination of both. For example, the left-to-right direction of axioms 3.18–3.22 should be represented as a deductive rule, whereas the right-to-left direction should be represented as an integrity rule.

The reader may wonder why these 31 axioms and axiom schemas have been chosen. The majority of the axioms involve the interpretation of three abstraction principles: classification (instantiation), generalization (specialization), and attribution. These three principles occur so frequently in modeling that predefining them makes sense. As an alternative, one can start with an empty list of axioms and regard all axioms as user-defined. The advantage of such an approach is an even greater flexibility at the expense of interference between domain-specific rules and domain-independent abstraction principles. For example, a user-defined rule could instantiate an individual object to the attribute object of `Proposition`. This would, however, destroy the intended interpretation of attribute and individual objects as links and nodes. Hence, the reason for having predefined axioms is to start from a well-understood set of abstraction principles that are protected, via axioms 3.30 and 3.31, against unwanted interference resulting from user-defined rules.

Section 3.13 presents a technique for defining more abstraction principles via metalevel formulas. In fact, a method engineer can define her own brand of attribu-

tion, instantiation, and specialization by creating metalevel formulas. Instead of using the labels `attribute`, `in`, and `isA`, she would define new attribute categories like `myAttribute`, `myIn`, and `myIsA`, then constrain their interpretation through user-defined rules.

3.9 User-Defined Constraints and Rules in Telos

The previous section introduced Telos as a framework for deductives databases with a single base relation (the P-facts) and some deduction rules and integrity constraints. Telos's three predicates, `(x in c)`, `(x m/l y)`, and `(c isA d)`, were defined in terms of the P-facts according to deductive rules. A number of predefined integrity constraints define which Telos databases are regarded as consistent. This section presents the logical language of Telos (as implemented in ConceptBase), which allows deductive rules and integrity constraints to be formulated at any abstraction level. ConceptBase provides a predefined object `Class`, which has two attributes, `rule` and `constraint`, that allow user-defined rules and constraints to be specified:

```
Class in Proposition with
  attribute
    rule: MSFOLrule;
    constraint: MSFOLconstraint
end
```

The acronym MSFOL in the preceding definition stands for *many-sorted first-order logic*: All variables are typed by a class name. The formulas are well-formed expressions over the predicates defined earlier. As a syntactic abbreviation, quantified variables are assigned to *class ranges*: `forall x/C F` is an abbreviation for `forall x (x in C) ==> F`, and `exists x/C F` is an abbreviation for `exists x (x in C) and F`. In ConceptBase, logical formulas are included between two dollar signs ($).

The example begun earlier in the chapter can now be continued. We assume that a *constraint* on the lower bound of salaries is formulated:

```
Employee in Class with
  constraint
    c1: $ forall e/Employee s/Integer
        (e salary s) ==> (s > 1500) $
end
```

The reader should be aware that Telos frames are incremental. The frame just presented has to be understood as additional information about the object `Employee`. More precisely, the object is made an instance of `Class`, and one constraint with label `c1` is added.[7]

Deductive rules are employed to derive new attribute relationships `(x m y)` or class memberships `(x in c)` from the database. As an example, we model `Department` as a class that can have subordinate departments. A deductive rule is used to define when a department has another department as its part:

source

```
Department in EntityType,Class with
  feature
    directSubordinate: Department;
    subordinate: Department
  rule
    r1: $ forall d1,d2/Department
        (d1 directSubordinate d2) ==> (d1 subordinate d2) $;
    r2: $ forall d1,d2/Department
        (exists d/Department (d1 directSubordinate d) and (d
        subordinate d2)) ==> (d1 subordinate d2) $
end
```

The two deductive rules effectively realize the transitive closure of the `direct-Subordinate` relation. The derived predicate `subordinate` can be used in other logical formulas; for example:

```
Department with
  constraint
    c1: $ forall d1,d2/Department
        (d1 subordinate d2) ==> not (d2 subordinate d1) $
end
```

Note that attribute labels are local to objects. Thus, the attribute `Employee!c1` is distinguished from the attribute `Department!c1`. The logical expressions in constraints and deductive rules can be arbitrarily nested using the logical operators `and`, `or`, `not`, `forall`, `exists`, and `==>`. The predicates must be correctly typed; that is, an attribute expression `(x m y)` or `(x m/l y)` is allowed only if the class of which `x` is an instantiation has an attribute with label `m`. For example, the constraint `Department!c1` is valid because the variable `d1` is assigned to class `Department`, which has an attribute `subordinate`.

3.10 Query Classes

The last and most flexible incarnation of logical expressions in ConceptBase is the *query classes*. A query class resembles an ordinary class with a constraint definition. The constraint, however, is interpreted as a sufficient condition for class membership: All instances that match the query class definition and fulfill the constraint are

regarded as instances of the query class. As an example, we first consider a class definition of `Department`, which has a `head` attribute and a constraint that each class must have a manager:

```
DepartmentWithBoss in EntityType,Class isA Department with
  feature
    head: Manager
  constraint
    c2: $ forall d/DepartmentWithBoss
        exists m/Manager (d head m) $
end
```

An attempt to create a query class with an instance without filling the `head` attribute (like `dept1`) will fail. The second instance (`dept2`) fulfills the constraint:

```
dept1 in DepartmentWithBoss
end

dept2 in DepartmentWithBoss with
  head
    manager: John
end
```

The idea of a query class is to let objects like `dept1` and `dept2` be instances of the superclass (`Department`) and to compute the membership of the query class:

source

```
Department with
  feature
    head: Manager
end

DepartmentWithBossQ in QueryClass isA Department with
  constraint
    c2: $ exists m/Manager (~this head m) $
end
```

The special variable `~this` denotes any instance object of the query class that fulfills the constraint. It can be regarded as an implicit universal quantification:[8]

```
DepartmentWithBossQ isA Department with
  constraint
    c1: $ forall this/Department
        exists m/Manager (this head m) $
end
```

The above frame appears in italics since it is not a valid Telos frame. It only shows how the variable "this" is interpreted.

In many cases, it is useful to compute those objects that violate a constraint like `DepartmentWithBoss!c2`. A query class with a negated constraint definition will output the violating objects. In our example, we are interested in those departments that do not have a head:

```
DepartmentWithoutBossQ in QueryClass isA Department with
  constraint
    c2: $ not exists m/Manager (~this head m) $
end
```

The classification of objects into query classes is a deductive computation. The class membership rule of `DepartmentWithoutBossQ` would be

```
forall this/Department
(not exists m/Manager (this head m)) ==> (this in
DepartmentWithoutBossQ)
```

ConceptBase generates such a rule internally in its implementation of query classes. Since the classification predicate `(x in c)` is used as the conclusion, a query class can be referred to like an ordinary class in other query classes or even inside itself. The following example shows a reference to a query class as a superclass and as a constant within the constraint:

```
TopSalary in QueryClass isA HighInteger with
  constraint
    c3: $ exists e/Employee (e salary ~this) and (~this >
        1000000) $
end

TopDepartmentQ in QueryClass isA DepartmentWithBossQ with
  constraint
    c2: $ exists m/Manager s/TopSalary (~this head m) and (m
        salary s) $
end
```

Query classes can be considered classes that offer a method `ask`: Whenever the method is called, those objects are returned that fulfill the constraint of the query class. Like other Telos classes, a query class can have more than one superclass:

```
Academic in EntityType end

AcademicEmployee in QueryClass isA Employee,Academic end
```

The interpretation of the preceding frame is that any instance of `AcademicEmployee` must be both an instance of `Employee` and `Academic`, or logically

```
forall x (x in Employee) and (x in Academic) ==> (x in
AcademicEmployee)
```

The rule is a means of *deducing* the instances of the query class. Such a deductive rule deriving class membership is generated for each query class. The logical representation allows query classes to appear anywhere that classes can appear. For example, the class `Academic` can itself be a query class. If a query class has some constraint, then its logical representation is added to the conjunction of the precondition:

```
RichAcademicEmployee in QueryClass isA AcademicEmployee with
  constraint
    c: $ exists s/TopSalary (~this salary s) $
end
```

The class membership rule is in this case:

```
forall x (x in AcademicEmployee) and (exists s/TopSalary (x salary
s)) ==> (x in RichAcademicEmployee)
```

ConceptBase compiles the class membership rule from the query class. If `Q` is a query class, then all objects `x` for which `(x in Q)` is true are called *answer objects* of `Q`, or simply instances of `Q`. Besides the simple class membership, a query class can also specify *answer attributes*. These are either attributes that the answer object has in the database (retrieved attributes, discussed in the next section) or they are attached to the answer object by a condition formulated in the query class constraint.

3.11 Attributes and Parameters in Queries

So far, the query classes examined have been capable only of expressing the instantiation of an object in their answer set. In practical application, one needs to retrieve properties of answer objects to query classes as well. Query classes provide two attribute categories for this. *Retrieved attributes* are attributes that an object has independent of the query class definition. *Computed attributes* are properties that are attached to the answer object via the query class definition.

The following definition shows a typical use of retrieved attributes:

source

```
RichAcademicEmployee1 in QueryClass isA AcademicEmployee with
  retrieved_attribute
    salary: Integer
  constraint
```

```
    c: $ exists s/TopSalary (~this salary s) $
end
```

ConceptBase generates a *query rule* for the retrieved attribute:

```
forall x,s (x in RichAcademicEmployee1) and (s in Integer) and
(x salary s) and ((s in TopSalary) and (x salary s)) ==>
Q(RichAcademicEmployee1,x,salary,s)
```

The class membership rule is derived from the following rule:

```
forall x,s Q(RichAcademicEmployee1,x,salary,s) ==> (x in
RichAcademicEmployee1)
```

A side effect of this representation is that, in the case of our example, an object `x` that has no salary (i.e., no value for the retrieved attribute) is not in the answer set of the query class. So retrieved attributes are *necessary*. They are not necessarily single-valued, however: An employee may, for example, have more than one salary. Both would be derived via the query rule.

The destination of a retrieved attribute can be a specialization of the attribute defined in a direct or an indirect superclass of `RichAcademicEmployee`. In our case, the salary attribute is defined for the class `Employee` and has the destination type `Integer`. Since we defined a subclass `TopSalary` of `Integer`, we can formulate a new version of the query:

```
RichAcademicEmployee2 in QueryClass isA AcademicEmployee with
  retrieved_attribute
    salary: TopSalary
end
```

This equivalent query definition no longer requires the constraint of the previous version. By coincidence, the class TopSalary is also a query class. Since the class membership rule for this query yields instantiations `(s in TopSalary)`, this is a correct and feasible way of referring to instances of query classes. The query rule is in this case

```
forall x,s (x in RichAcademicEmployee2) and (s in TopSalary) and
(x salary s) ==> Q(RichAcademicEmployee2,x,salary,s)
```

The reader can verify that this version of the query rule is logically equivalent to the previous version.

In general, a query class can have any number of retrieved attributes. The use of retrieved attributes can be compared to a projection in classical relational algebra: Only the required attributes are returned in the answer to the query class. The differ-

ence between the use of retrieved attributes and relational algebra is that retrieved attributes can be defined in any superclass of the query class. They may also be derived by means of a deductive rule:

```
DepartmentWithBossQ1 in QueryClass isA Department with
  retrieved_attribute
    head: Manager;
    subordinate: Department
  constraint
    c2: $ exists m/Manager (~this head m) and not (m head
        ~subordinate) $
end
```

The query returns departments (along with their heads and subordinate departments) in which the head of the department is not head of the subordinate department.

The second type of answer attribute, the computed attribute, is derived within the query constraint. As an example, consider the `head` attribute of `Department`. Instead of attaching the `head` in `DepartmentWithBossQ1` to departments, one can formulate a query that returns instances of `Manager` together with the departments of which the managers are head:

```
BossWithDeptQ in QueryClass isA Manager with
  computed_attribute
    dept: Department
  constraint
    c2: $ (~dept head ~this) and not (exists upper/Department
        (upper subordinate ~dept)) $
end
```

The second part of the constraint in this example makes the query class return only those managers in its answer who head a department that is not subordinate to an upper-level department. Only those managers will be returned that are heads of at least one department, because the expression `~dept` stands for an existentially quantified variable in the constraint:

```
exists ~dept/Department (~dept head ~this) and not (exists upper/
Department (upper subordinate ~dept))
```

The construct for computed attributes is per se redundant. Deductive rules have the expressive power to deduce these attributes as well. However, deductive rules are not allowed in a definition of a query class, only in that of an ordinary class. I show the equivalent deductive rule here for the sake of clarification:

```
Manager in Class with
  attribute
    dept: Department
  rule
    deptrule: $ forall m/Manager d/Department (d head m) ==> (m
              dept d) $
end
```

Note that the `dept` attribute does not use the attribute category `feature` but rather the predefined category `attribute` that is available to all Telos objects. Moreover, the object `Manager` is classified into two classes: `EntityType` (see section 3.7) and `Class`. The latter provides the attribute category `rule`, which is instantiated by `deptrule`, more precisely, `(Manager!deptrule in Class!rule)`. Assuming that `deptrule` is defined, the query class is expressed as follows:

```
BossWithDeptQ1 in QueryClass isA Manager with
  retrieved_attribute
    dept: Department
  constraint
    c2: $ (~this dept ~dept) and not (exists upper/Department
        (upper subordinate ~dept)) $
end
```

The solution with the computed attribute is preferable here, since it does not require extending a class by an attribute plus a deductive rule.

The final remaining feature for query classes is parameterization. A *generic query class* can contain parameter definitions. Logically, a parameter definition introduces an existentially quantified variable in the query constraint. When calling a query, a user can supply values for parameters (*parameter instantiation*) or restrict a parameter to some subclass of its original range (*parameter specialization*).

As an example, consider the definition of the class `EntityType` provided at the beginning of section 3.7. It mentions an attribute `feature` with `Domain` as value. Assume that we are interested in getting the list of all instances of `EntityType` that have some instance of `Domain` as `feature`. The filler for `whatDomain` will be provided not at query definition time, but at query call time. The generic query class is then

```
EntityTypeByDomainQ in GenericQueryClass isA EntityType with
  parameter
    whatDomain: Domain
  constraint
```

```
    c2: $ (~this feature ~whatDomain) $
end
```

The constraint ensures that only those instances of `EntityType` are returned that have a feature matching the parameter `whatDomain`. The parameter is a shortcut for a constraint formula that has an existentially quantified variable for the parameter:

```
exists ~whatDomain/Domain (~this feature ~whatDomain)
```

The query rule for a generic query class is built like that for an ordinary query class. The parameter becomes a variable in the conclusion predicate:

```
forall x,w (x in EntityType) and (w in Domain) and (x feature w)
==> Q(EntityByDomainQ,x,whatDomain,w)
```

Remark 3.2 (on quantification) The query rule has a universal quantification for the parameter variable `w`. This is consistent with the existential quantification of the parameter `~whatDomain` in the constraint, since the constraint is part of the condition of the rule. Consider the general example of a deductive rule:

```
forall x,y A(x,y) ==> B(x)
```

This is equivalent to the version in which the `exists` is pushed into the condition:

```
forall x (exists y A(x,y)) ==> B(x)
```

If a user wants to replace a parameter by an instance (parameter instantiation), she calls a query expression of the form

```
QC[parameter-subst1,...,parameter-substn]
```

where `QC` is the name of the generic query class, and the parameter substitutions are enclosed in square brackets. A parameter substitution for instantiation has the form

```
v/p
```

meaning that the query parameter `p` is replaced with the value `v`. In our example, the query call

```
EntityTypeByDomainQ[Integer/whatDomain]
```

returns all instances of `EntityType` that have some feature with domain `Integer`, e.g., `Employee`. A parameter substitution replaces the parameter variable `p` with the constant value v supplied in the query call. The substitution is applied to the conclusion predicate of the query and instantiates all matching parameter variables. In the running example, the substitution for parameter variable `w` (`whatDomain`) yields the substituted query rule

```
forall x (x in EntityType) and (Integer in Domain) and (x feature
Integer) ==> Q(EntityByDomainQ,x,whatDomain,Integer)
```

A query class may have more than one parameter. A query call may provide parameter substitutions for all or some of the parameters defined for the query. The parameters that are not substituted remain existentially quantified.

A second type of parameter substitution involves *specializing* the parameter. Each parameter has a range associated with it by the query class definition (for example `Domain` is the range of the parameter `whatDomain`). When calling the query, the user can provide a stricter range, that is, a subclass of the original range:

```
NumberDomain in Node isA Domain end
Integer in NumberDomain end
Real in NumberDomain end
```

Here, `NumberDomain` is a subclass of `Domain` and thus a possible stricter range for the parameter `whatDomain`. The syntax for parameter specialization is

```
p:C
```

where `C` is the name of a subclass of the original range of the parameter `p`. A parameter specialization replaces the range of the parameter's variable in the query rule. For example, the query call

```
EntityTypeByDomainQ[whatDomain:NumberDomain]
```

is evaluated on the substituted query rule

```
forall x,w (x in EntityType) and (w in NumberDomain) and (x
feature w) ==> Q(EntityByDomainQ,x,whatDomain,w)
```

If a generic query class contains more than one parameter, then a query call can contain any mixture of parameter instantiations and specializations for all or part of the query's parameters. One parameter may not, however, be both instantiated and specialized in the same query call.

The attribute category `parameter` is defined for generic query classes. It may be combined with the categories `retrieved_attribute` and `computed_attribute`. For example, the query class

```
EntityTypeByDomainQ1 in GenericQueryClass isA EntityType with
  parameter, retrieved_attribute
    feature: Domain
  constraint
    c2: $ (~this feature ~feature) $
end
```

returns as answer attribute the feature(s) of instances of `EntityType`. At the same time, the user can parameterize this attribute. The query call expression

```
EntityTypeByDomainQ1[Integer/feature]
```

returns all instances of `EntityType` that have at least one feature of domain `Integer` and would return, in the answer, the features of `Integer` that the instances have.

In principle, a query call is allowed at any position in a Telos frame where a Telos class is allowed. ConceptBase however, allows query calls only inside Telos frames of ordinary classes.

3.12 Views as Extended Query Classes

Views extend the functionality of generic query classes in two directions. First, they allow complex attributes. Second, they introduce a variant of retrieved attributes that allows answers to have fillers (or not) on specified attributes. The following definition of the class `View` introduces the new features:

```
View in Class isA GenericQueryClass with
  attribute
    inherited_attribute : Proposition;
    partof : SubView
end
```

Attributes of the category `inherited_attribute` are similar to retrieved attributes of query classes, but they are not necessary for answer objects of the views; that is, an object need not necessarily instantiate the attribute to be a solution of the view.

The `partof` attribute in the class `View` allows the definition of complex nested views; that is, attribute values are not restricted to simple object names. They can also represent complex objects with some further attributes. The following view retrieves all employees with their departments and attaches the head attribute to the departments:

```
EmpDept in View isA Employee with
  retrieved_attribute, partof
    dept : Department with
               retrieved_attribute
                    head : Manager
           end
end
```

The subview at `EmpDept!dept` is an abbreviated view definition in which the attribute value (here, `Department`) has the role of the subview's superclass. ConceptBase internally decomposes the nested view definition into several nonnested view definitions:

```
EmpDept in View isA Employee with
  retrieved_attribute, partof
    dept : SV_EmpDept_dept
end

SV_EmpDept_dept in SubView isA Department with
  retrieved_attribute
    head : Manager
end
```

A view definition can have arbitrarily many subviews. Each subview can itself have subviews. The level of nesting in the view definition determines the nesting within the answer objects. In frame syntax, subviews appear as follows:

```
John in EmpDept with
  dept
    JohnsDept : Production with
      head
        ProdHead : Phil
    end
end

Max in EmpDept with
  dept
    MaxsDept : Research with
      head
        ResHead : Mary
    end
end
```

If one replaces `retrieved_attribute` with `inherited_attribute` in the view definition, then the corresponding attribute may have zero fillers in the answer that is returned. As an example, consider the following view definition:

```
EmpDept1 in View isA Employee with
  retrieved_attribute, partof
    dept : Department with
               inherited_attribute
                    head : Manager
           end
end
```

An employee like `Mary` whose department has no head might also be in the answer set of the view:

```
Mary in EmpDept1 with
  dept
    MaxsDept : Marketing
end
```

To make it easier to define views, we allow some shortcuts in the view definition for the classes of attributes. For example, if one wants all employees who work in the same departments as `John`, one can use the term `John.dept` instead of `Department`. In general, the term `object.attrcat` refers to the set of attribute values of `object` under the attribute category `attrcat`, that is, all objects `x` such that `(object attrcat x)` holds. This path expression can be extended to any length; for example, `John.dept.head` refers to all managers of departments in which John is working.

A second shortcut permitted in view definitions is the explicit enumeration of allowed attribute values. The following view retrieves all employees who work in the same department as `John` and earn 10000, 15000, or 20000 euros.

```
EmpDept2 in View isA Employee with
  retrieved_attribute
    dept : John.dept;
    salary : [10000,15000,20000]
end
```

As mentioned before, subviews use the same syntax as normal view definitions. Constraints can also be specified in subviews; such constraints refer to the object of the outer frame(s).

```
EmpDept_likes_head in View isA Employee with
  retrieved_attribute,partof
    dept : Department with
        retrieved_attribute, partof
            head : Manager with
                constraint c : $ (~this likes ~this::dept::head) $
        end
    end
end
```

The variable `this` in nested views always refers to the object of the main view, in this case, an employee. Objects of nested views can be referred to using `~this::label`, where `label` is the corresponding attribute name of the subview. In the example, we want to express the idea that employees must like their bosses. Because the subview for managers is part of the subview for departments, we must use the :: operator

twice: `~this::dept` refers to the departments of `~this`, and `~this::dept::head` refers to the heads of the departments of `~this`.

3.13 Metalevel Formulas

Deductive rules and integrity constraints are formulas over the set of allowed predicates. In our example, the formulas are attached to classes and make statements about the instances of the classes. For method engineering, one has to design modeling notations in which the symbols like those for nodes and links have a predefined semantics: When a model is created using these symbols, then their interpretation has to be consistent with the definition of the modeling notation. For method engineering, the formulas range not just over instances of classes, but over instances of instances of classes. As an example, consider the subclass relationship. In Telos, this is encoded with the `isA` predicate. Its interpretation is defined by the Telos axioms, namely,

```
forall o (o in Proposition) ==> (o isA o)
forall c,d,e (c isA d) and (d isA e) ==> (c isA e)
forall x,o,c,d (x in c) and P(o,c,isa,d) ==> (x in d)
```

The axiom of interest here is the last one. It quantifies over variables `c`, `d`, which themselves are classes that stand in subclass relationship to each other.

Definition 4.1 Let `f` be a Datalog formula. `f` is called a *metalevel formula* iff a predicate `(x in c)` with variable `c` occurs in `f`.

We say that `c` occurs at the class position of the predicate `(x in c)`.

As such, metalevel formulas are nothing special. They do, however, have serious implications for stratification in a Telos database: About half of all Telos objects are instantiation objects. If we stratified on the predicate `in` for instantiation, then we would not be allowed to use a negative literal `not (x in c)` in any rule that directly or indirectly derives the predicate `(x in c)`. Because the `in` predicate is so frequently used in Telos, this would basically eradicate the use of negation. To overcome this problem, ConceptBase applies stratification to the predicate symbol `In.c`, that is, the combination of the predicate name plus the second argument. This method, however, doesn't work directly for metalevel formulas, since they feature `in` predicates with a variable as second argument.

The problem can be overcome by *partial evaluation*. If the metalevel formula `f` contains a positive literal `Q(...,c,...)`, which binds the variable `c`, then we compute the extension of `Q` and rewrite the metalevel formula with all facts from the extension of `Q`. In the case of the `isA` rule, we can take the predicate `P(o,c,isa,d)` in the role of the predicate Q. Assume that we have two facts `P(o1,Emp,isa,Person)`

and `P(o2,Manager,isa,Emp)` in the extension of the P-predicate. Then the metalevel formula is replaced by two partially evaluated formulas:

```
forall x (x in Emp) ==> (x in Person)
forall x (x in Manager) ==> (x in Emp)
```

The partially evaluated formulas now have constants as second arguments of the `in` predicates, yielding better opportunities for stratification.

The partial-evaluation algorithm for metalevel formulas is more complex than the foregoing example suggests. In general, there can be more than one Q-predicate for use in partial evaluation. The system then has to select one that does not have too many facts in its extension. (A huge extension would result in a huge number of partially evaluated formulas.) In some cases, the partial evaluation must be applied more than once to eliminate all variables at class positions of `in` predicates.

Example 3.1 As an example for metalevel formulas, consider cardinality of attributes. Let's assume that we want to specify that certain attributes require a filler (`necessary`) or have at most one filler (`single`):

```
Manager in Class with
  attribute
    dept: Department;
    leads: Project
  constraint
    onDept: $ forall m/Manager d1,d2/Department
            (m dept d1) and (m dept d2) ==> (d1 == d2) $;
    onLead: $ forall m/Manager
            exists p/Project (m leads p) $
end
```

Obviously, the `dept` attribute is single-valued and the `leads` attribute is necessary. However, the constraints are formulated at the model level (here, with `Manager`). We require a generic formulation of single and necessary attributes that we can reuse for any class. To obtain such a generic formulation, we extend the class `Proposition` with two more attributes and define two metalevel formulas:

source

```
Proposition in Class with
  attribute
    single: Proposition;
    necessary: Proposition
  constraint
    singleIC: $ forall p/Proposition!single c,d/Class x,m/VAR
    P(p,c,m,d) and (x in c) ==> (forall y1,y2/VAR (y1 in d) and
    (y2 in d) and (x m y1) and (x m y2) ==> (y1 == y2)) $;
```

```
    necessaryIC: $ forall p/Proposition!necessary c,d/Class
    x,m/VAR
    P(p,c,m,d) and (x in c) ==> (exists y/VAR (y in d) and (x m
    y)) $
end
```

The pseudoclass range `VAR` can be used when the corresponding variable gets a range from the partial evaluation or is replaced by a constant. For example, the variable `x` gets a range as a result of the partial evaluation of predicate `P(p,c,m,d)`. The same predicate replaces the variable `m` with constants. The metaformulas for `single` and `necessary` allow their semantics to be reused simply by using the appropriate attribute categories:

```
Manager in Class with
  single
    dept: Department
  necessary
    lead: Project
end
```

The partial evaluation for the `single` attribute works as follows: The database implies the facts `(Manager!dept in Proposition!single)` and `P(Manager!dept, Manager,dept,Department)`. This matches the conjunction `(p in Proposition! single)` and `P(p,c,m,d)`[9] in `singleIC`, leading to a substitution:

```
[Manager!dept/p, Manager/c, dept/c, Department/d]
```

Applying this substitution to the `singleIC` formula yields the partially evaluated formula

```
forall x/VAR (x in Manager) ==> (forall y1,y2/VAR (y1 in
Department) and (y2 in Department) and (x dept y1) and (x dept y2)
==> (y1 == y2))
```

which can be rewritten as

```
forall x/Manager y1,y2/Department (x dept y1) and (x dept y2) ==>
(y1 == y2)
```

Figure 3.4 illustrates that metalevel formulas are defined at the level of metaclasses. These formulas define properties of the instances of classes, which are themselves instances of the metaclasses. Hence, if we use the metaclass level for defining modeling notations, metalevel formulas can be employed to define part of the semantics of the notational symbols (in our example, the semantics of the `single` and `necessary` symbols). Note that the class `Proposition` serves as a metaclass. Since any object is an instance of `Proposition`, this is a consistent use of classification. In

gel

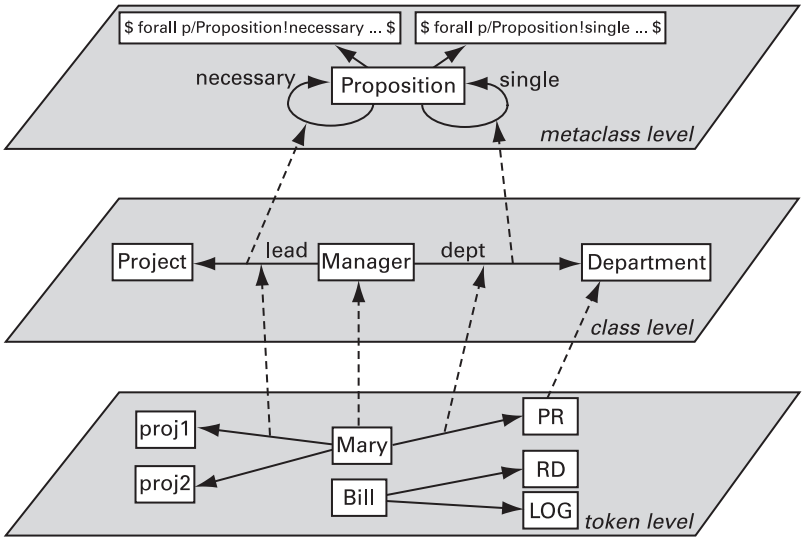


Figure 3.4
Metalevel formulas as properties of metaclasses

the figure, the token object `Mary` is a proper instance of `Manager` because neither the `singleIC` nor the `necessaryIC` is violated. The object `Bill`, on the other hand, violates both constraints.

Example 3.2 This example is about defining the meaning of *transitivity*, *symmetry*, and similar relation properties. These properties are so basic that they are part of any elementary textbook in algebra. Hence, it would be useful to have predefined attribute categories for them rather than formulating transitivity over and over again for specific relations like the `subordinate` attribute of section 3.9. The following definitions solve the problem at the most generic level, the level of `Proposition`:

```
Proposition with
  attribute
    transitive: Proposition;
    antisymmetric: Proposition
end
```

The two metalevel formulas for transitivity and antisymmetry are then defined by

source

```
RelationSemantics in Class with
  rule
    trans_R: $ forall x,z/VAR
             (exists AC/Proposition!transitive C/Proposition y/VAR
```

```
            M/VAR P(AC,C,M,C) and (x in C) and (y in C) and (z in
            C) and (x M y) and (y M z)) ==> (x M z) $
  constraint
    antis_IC: $ forall AC/Proposition!antisymmetric C/Proposition
              x,y/VAR M/VAR
              P(AC,C,M,C) and (x in C) and (y in C) and (x M y)
              ==> not (y M x) $
end
```

The reader will recognize the pattern in the last lines of the two formulas; the final clauses in each formula above define the scope of the variables. Because transitivity and antisymmetry are defined at the level of the most generic object, `Proposition`, these attribute categories can be applied to any attribute definition just by attaching them as attribute categories. Consider a class `Person` that has an ancestor attribute that should be antisymmetric and transitive. Given the two preceding formulas, the attribute can be fully specified by

```
Person with
  attribute,antisymmetric,transitive
    hasAncestor: Person
end
```

The properties symmetry and reflexivity can be defined by similar expressions. The full details are in the example models provided on the CD-ROM that accompanies the volume.

ConceptBase supports metalevel formulas for rules and constraints of proper classes, but not for query classes. Although metalevel formulas do not extend the formal expressiveness of Telos, they save a lot of coding effort, and they can be seamlessly integrated into the IRDS-based strategy for defining modeling notations. The more metalevel formulas are expressed at the metaclass level, the deeper is the explicit knowledge about the modeling notations, in particular, the meaning of the symbols in the modeling notations. Section 3.15.6 comes back to this issue to define the meaning of cardinality expressions in data-modeling notations. Chapter 7 elaborates on an even more complex application of metalevel formulas.

3.14 Active Rules

Rules, constraints, and queries are the principal means of expressing computation in the ConceptBase metadatabase system. Since they are all mapped to Datalog with negation, one can guarantee termination of any computation. Any computation beyond the Datalog scope is supposed to take place in application programs that interact with the metadatabase system.

In order to extend the system's computational scope, *active rules* have been introduced in ConceptBase. An active rule has three parts. The first part, the *event section*, specifies what external event activates the active rule. With ConceptBase, an external event can be an update to the database or a calling of a query.[10] An event section binds variables to values. The second part is the *condition section*, a logical formula over a superset of the variables bound by the event section. This formula is evaluated over the database state. The last part is the *action section*. It consists of a sequence of procedural calls that either update the database (insert, delete) or invoke further query calls, including built-in queries (queries without a logical constraint that execute a piece of program code).

The class definition of active rules is as follows:

```
ECArule in Class with
  attribute
    ecarule : ECAassertion;
    priority_after : ECArule;
    priority_before : ECArule;
    mode : ECAmode;
    active : Boolean;
    depth : Integer
end
```

The attribute `ecarule` holds the specifications for the event, condition and action parts (sometimes referred to by the acronym ECA). An ECA rule is represented as a string that starts with variable declarations:

```
$ v1/c1 v2/c2 ...
ON event
IF condition
DO actions-1
ELSE actions-2 $
```

The first line in this example contains the declaration of all variables used in the `ECAassertion`. The specified classes of the variables are used only for compilation of the rule; during the evaluation of the rule, whether the variables are instances of the specified classes is *not* tested. The variables are bound to objects by event, condition, or action statements.

Possible events are the insertion (`Tell`) or deletion (`Untell`) of attributes (predicate `A`), instantiation links (predicate `In`), or specialization links (predicate `Isa`). For example, if the rule should be executed if an object is inserted as instance of `Manager`, then the event statement is

```
Tell(In(x,Manager))
```

Furthermore, an event may be a query; for example, if the event

```
Ask(find_instances[Employee/class])
```

is specified, the ECA rule is executed before the evaluation of the query `find_instances` with the parameter `Employee`. It is possible to use a variable as a placeholder for a parameter.

The event detection algorithm takes only extensional events into account. Events that can be deduced using a rule or a query are not detected. However, the algorithm cares about the predefined Telos axioms; for example, if an object is declared as an instance of a class, the object is also an instance of the superclasses of that class.

The condition section of an active rule is a predicate evaluated on the database. It can be either a normal literal (`A`, `In`, or `Isa`) or a query like `In(x,EmpDept1)`. If the condition contains a free variable, the actions specified in the `DO` block are executed for each result for this variable. If the condition contains only constants or bound variables and can be evaluated to TRUE, the `DO` action block is executed once. Otherwise, the `ELSE` block is executed. By default, queries are evaluated on the state of the object base before the database transaction started. If one wants to take new information into account in the evaluation of a query, one has to use the `new` operator. For example, if one wants to check the instantiation of an object in the database state after the effect of update operations, one has to specify

```
new(In(x,Person)
```

in the condition.

Actions are specified in an active rule in a comma-separated list. The syntax is similar to that of events, except that one can also pose queries (`Ask`). All variables in `Tell` and `Untell` actions must be bound. The `Tell` action of an attribute `A(x,ml,y)` is performed only if there is no attribute of category `ml` with value `y` for object `x`. In that case, a new attribute with a system-generated label is created. If an attribute `A(x,ml,y)` is to be deleted, then all attributes of category `ml` with value `y` for object `x` are deleted.

Active rules are linked to transactions on the database. A *transaction* is a sequence of updates (insertions or deletions) and queries. All events in active rules are derived from the entries in the transaction. The attribute `mode` controls when an active rule is evaluated. It has three possible modes:

- **Immediate:** The condition in the rule is evaluated immediately after the event specified in the rule has been detected. If it evaluates to TRUE, the action is executed immediately, too.

• **ImmediateDeferred:** The condition in the rule is evaluated immediately after the event specified in the rule has been detected. If it evaluates to TRUE, the action is executed at the end of the current transaction.[11]

• **Deferred:** The condition in the rule is evaluated at the end of the current transaction. If it evaluates to TRUE, the action is executed immediately after the evaluation of the condition.

The default mode is `Immediate`.

The attributes `priority_after` and `priority_before` establish a partial order on the set of active rules. If more than one active rule has a particular event as its trigger, the partial order determines the sequence in which the active rules are evaluated. The attribute `active` has possible values TRUE and FALSE. It allows an active rule to be deactivated/reactivated without its having to be deleted from or reinserted into the database.

Finally, the attribute `depth` controls the depth of the call tree of active rules: The action part of an active rule can produce a new event, which triggers the call of another active rule, which itself can trigger the call of further active rules. When the nesting depth specified in `depth` is reached, the execution of nested active rules is aborted. This feature prevents infinite loops. The default value for the depth is zero; that is, no nested calls are permitted.

Example 3.3 We model a situation in which employee candidates are entered into a database. Those candidates fulfilling a certain exception condition are put on a "watch list" (first ECA rule). Otherwise they are entered as ordinary employees into the system. Those candidates placed on the watch list are removed and entered as normal employee as soon as the system acknowledges that it has "watched" them.

First, the necessary classes have to be defined. The query class `UnderPayed` defines an exception employee candidate:

source

```
EmployeeCandidate in Class with
  attribute
    salary: Integer
end

Employee isA Employee end
ToBeWatched end
AlreadyWatched end

UnderPayed in QueryClass isA EmployeeCandidate with
  constraint
    c1: $ exists s/Integer (~this salary s) and (s < 1000) $
end
```

The first ECA rule is triggered whenever an instance of `EmployeeCandidate` is entered into the database. When it is triggered, the exception condition `UnderPayed` is checked for that candidate on the new database state. If the condition is TRUE, the candidate is placed on the watch list; otherwise she is entered into the database as a normal employee:

```
WatchForUnderpayed in ECArule with
  mode m: Deferred
  ecarule
    er : $ e/Employee
      ON Tell(In(e,EmployeeCandidate))
      IF new(In(e,UnderPayed))
      DO
        Tell(In(e,ToBeWatched))
      ELSE
        Tell(In(e,Employee))
      $
end
```

The second ECA rule is for removing a candidate from the watch list. It is triggered by an instantiation of the class `AlreadyWatched`:

```
Okayed in ECArule with
  mode m: Deferred
  ecarule
    er : $ e/EmployeeCandidate
      ON Tell(In(e,AlreadyWatched))
      IF TRUE
      DO
        Untell(In(e,ToBeWatched)),
        Tell(In(e,Employee))
      $
end
```

The second ECA rule has no ELSE part, as when the condition is false, no action is executed. Note that the action part in this example contains two actions.

The example can be executed by inserting the following frames.

```
mary in EmployeeCandidate with
  salary s: 500
end

mary AlreadyWatched end
```

3.15 Engineering the Yourdan Method

Modern Structured Analysis (Yourdan 1989) is an example of a collection of modeling languages that is comprehensive while still rather simple. It was proposed before the shift in modeling to object orientation. The purpose of the Modern Structured Analysis method (referred to here as the Yourdan method, after its originator) is to specify a system in a semiformal way through graphical notations. A *system* here is an object that receives information from the environment and reacts to it by generating output or changing its internal state. Yourdan's method serves here as a test case to demonstrate the metamodeling approach with Telos.

Developers of system-modeling methods have observed that a system can be viewed from multiple perspectives. First, one represents functions (or processes) of the system with their inputs and outputs. This is the *functional perspective*. Second, the *data perspective* specifies about which entities of the external world the system makes records and determines the structure of the data elements processed by the system. Third, the *control perspective* augments the other two perspectives, defining how the system reacts to control events triggered by some object (or person) from the environment.

This section discusses the engineering of the Yourdan method in order to highlight the various aspects of method engineering and metamodeling. The presentation starts with the data perspective, because of its relative simplicity. A very simple notation definition level is assumed that is then applied to the functional perspective. We then discuss the development of internotational rules and constraints, which relate models developed in different perspectives. Finally, process models are presented that encode the steps of a modeling method (rather than the results of the steps, which are encoded in the modeling notations). The Telos formalizations of the Yourdan method are contained in the companion CD-ROM to this volume. They can be tested directly with the ConceptBase system. The Yourdan method case study presented in this section has the following structure:

sources
slides

- Modeling the ERD-simple notation: This is a basic version of the entity-relationship diagramming notation. It features entity types, relationship types, and attributes.
- Modeling the ERD-advanced notation: This notation extends the previous one with IsA relations and cardinalities.
- Modeling the DFD notation: This notation encodes data flow diagrams.
- Modeling internotational constraints: Constraints and queries can be used to manage the connections between models for different perspectives of the same artifact. The generation of these constraints can be managed via a richer notation definition level.

• Modeling process model notations: The individual steps of a method can themselves be encoded in a notation.

3.15.1 Modeling the ERD-Simple Notation

Figure 3.2 introduced abstraction levels that can be used to define modeling notations as well as models, and their incarnations. Entity-relationship diagrams (ERDs) are models for specifying data sets. *Entity types* are the denotations of sets of identifiable objects (entities) that can have descriptive properties (attributes). An attribute has a label. Its values are elements from a domain, for example, `integer` or `string`. A *relationship type* denotes a set of relationships. A relationship links several participating entities. The relationship type defines *roles* in which entities can participate. For example, a product entity can be linked to a customer entity by a relationship of type `orders`.

Relationship types can restrict the cardinalities of the entities that participate in a relationship. For example, in an `orders` relationship there must be exactly one customer and at least one product. The Telos model of ERD notation first defines the base concepts `Node` and its attribute `connectedTo` mentioned in the foregoing:

source

Notation definition level

```
Node with
  attribute
    connectedTo: Node
end
```

This simple notation definition level just provides the concepts `Node` and `Node!connectedTo`, that is, the ability to represent graphs. It is sufficient for the moment. We will later extend it to represent software development steps.

Notation level for a simple ERD notation

```
ObjectType in Node end {* used later *}

EntityType in Node isA ObjectType with
  connectedTo
    ent_attr: Domain
end

RelationshipType in Node isA ObjectType with
  connectedTo
    role: EntityType
end

Domain in Node end
```

Model level for an example ERD

```
Integer in Domain end
Domain String end
Date in Domain end
String isA Date end

Employee in EntityType with
  ent_attr
    e_name: String;
    earns: Integer
end

Project in EntityType with
  ent_attr
    budget: Integer
end

worksFor in RelationshipType with
  role
    toEmp: Employee;
    toProj: Project
end
```

The first three objects in this model define the three domains `Integer`, `String`, and `Date`. The classes `Integer` and `String` are predefined in ConceptBase. The class `Date` is defined here to subsume `String`. In other words, any instance of `String` is allowed as an instance for `Date`. This is a technical trick for use when the details of a particular domain are not of further interest in the modeling. The domains are part of the model level because their instances (the values) are at the data level.

The remainder of the model defines two entity types and one relationship type. Note that the relationship type uses the role names `toEmp` and `toProj`. Since they point to different entity types, these role names are not displayed in an ERD. However, the Telos representation has to assign such names because the role links have to be identifiable. An equivalent definition would be

```
worksFor in RelationshipType with
  role
    role1: Employee;
    role2: Project
end
```

Figure 3.5 displays the model in a conventional ERD and contrasts it with the Telos representation. The difference is that the Telos representation uniformly

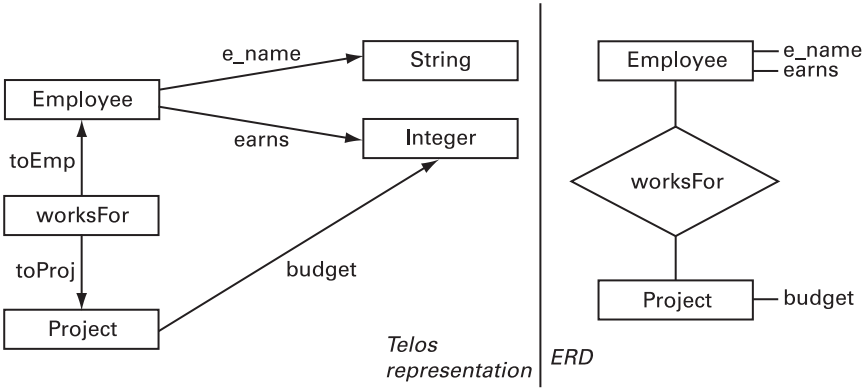


Figure 3.5
ERD versus its Telos representation

assigns role names and attaches domains to attributes. The ERD has different graphical symbol for entity types and relationship type, unlike the Telos representation, which uses the same symbol for both. The Telos model includes this information via the classification links `(Employee in EntityType)`, `(Project in EntityType)`, and `(worksFor in RelationshipType)`. The role links in the ERD appear undirected, but they always link a relationship type with an entity type. Therefore, the role links are implicitly directed. It follows that the ERD can always be reconstructed from the Telos representation.

The ERD-simple notation has no user-defined constraints (called *balancing rules* in the Yourdan textbook) and lacks constructs for IsA relationship types and complex object types (called *associative object indicators* by Yourdan). The following section introduces these two constructs, and the CD-ROM companion to this volume gives extensive examples of how to model constraints on ERD models and their semantics.

3.15.2 Modeling the ERD-Advanced Notation

source

In addition to the properties introduced in the ERD-simple notation, we might want the following features:

1. There should be a special relationship type IsA that has one entity type as "supertype" and several entity types as "subtypes." There can be several IsA relationship types that have a given entity type as supertype. Similarly, an entity type may occur as subtype in multiple IsA relationship types. The intended semantics are that each instance of a subtype is also an instance of the supertype of which it is a subtype.

2. A new object type should be introduced that is both an entity type and a relationship type (sometimes called a complex object type). The classical example is an order

that has role links to several entity types but also has some attributes and may be referred to by other relationship types.

3. Cardinalities for role links should be introduced. (We will limit consideration here to the cardinalities 1, 2, and many.) Cardinalities need to be enforced at the data(base) level and must be used in a consistent manner. For example, one may not specify that a role link is both "minimum 2 fillers" and "maximum 1 filler."

4. Key attributes should identify entities; that is, there should not be two different entities with the same value for the key attribute(s).

Models for the ERD-advanced notation are included on the companion CD-ROM to this volume.

3.15.3 Modeling the DFD and Event List Notations

Data flow diagrams consist of processes, data stores, data flows, terminators, control processes, and control flows. The DFD notation employed here has a new element, *leveling*, a way to decompose a process into a DFD consisting of DFD elements (processes, stores, etc.). The processes in the leveled DFD can themselves be leveled or specified by a process specification (pseudocode).

Since we are now defining a second notation, it makes sense to introduce a slightly extended notation definition level that allows us to keep models separate from one another and to express the idea that certain models belong to the same modeling project.

Notation definition level with `Model` and `Project`

```
Node in Individual with
  attribute
    connectedTo: Node
end

Project in Individual with
  attribute
    produces: Model
end

Model in Individual with
  attribute
    contains: Node
end
```

The new metaconcept `Model` aggregates content (here, nodes) and can be compared with a file containing records. The metaconcept `Project` aggregates several models

that belong to the same modeling project. Among other things, the two new concepts at notation definition level allow the expression of constraints on models that depend on whether they belong to the same modeling project.

Notation level for DFD notation

source

slides

```
Node DFD_Node in Node with
  connectedTo
    dataflow: DFD_Node
end

DFD_Node!dataflow with
  attribute
    withType: DataType
end

Process in Node isA DFD_Node with
  connectedTo
    leveledTo: DFD_Figure
end

Terminator in Node isA DFD_Node with
end

Store in Node isA DFD_Node with
  connectedTo
    withType: ObjectType
end

DataType in Node with
end

DFD_Model in Model with
  connectedTo
    contains: DFD_Node
end

DFD_Figure in Node,Model isA DFD_Model end

nowhere in DFD_Node end

PreliminaryBehavioralModel isA DFD_Model end

DataType in Node with
end {* reference to data dictionary *}
```

```
ObjectType in Node isA DataType with
end {* reference to ERD notation *}
```

The notation level for DFD makes use of the fact that some model components (`DFD_Node`) can be aggregated with the model (`DFD_Model`). The leveling relates a process to a `DFD_Figure` (a special DFD model). The leveled `DFD_Figure` can contain data flow links whose source or destination is not part of the `DFD_Figure`. Such data flows are called *dangling links* in the Yourdan method. The object `nowhere` serves as an artificial source and destination for those data flows. The concept `PreliminaryBehavioralModel` refers to a DFD model that is created from the event list (see later discussion).

The DFD notation also includes the notion of a *control process*. A control process has no dataflows but does have *control flows*. Incoming control flows are labeled by control events denoting that some condition in the environment or in the system has become TRUE. Outgoing control flows have no label and always connect a control process with an ordinary process: The control process can activate the process under certain conditions. The additional definitions for control processes are as follows:

```
DFD_Node in Node with
  connectedTo
    incomingCF: ControlProcess
end
DFD_Node!incomingCF with
  attribute
    withEvent: EventType
end

ControlProcess in Node isA DFD_Node with
  connectedTo
    outgoingCF: Process
end
```

Event types specify the nature of the incoming control flow. Some event types originate in the environment and are grouped in an *event list*. These external event types can be associated with a terminator, which originates the event. We distinguish three subclasses of external event types: *Flow event types* are defined as data inputs from a terminator to the system; *control event types* state that a certain condition has become TRUE in the environment; and *temporal event types* are TRUE when a certain point in time has been reached. (The precise definition of these event types can be found in textbooks about the Yourdan method.) It is worth noting that the models presented here group some details together: here an event list is a just a list of external event types.

Notation level for event list notation

```
EventType in Node with
  attribute
    eventtext: String
end

ExternalEventType in Node isA EventType with
  connectedTo
    agent: Terminator;
    answeredBy: Process {* links events to processes *}
end

EventList in Model with
  contains
    containsEvent: ExternalEventType
end

FlowEventType in Node isA ExternalEventType end
ControlEventType in Node isA ExternalEventType end
TemporalEventType in Node isA ExternalEventType end
```

Figure 3.6 depicts DFD notation and its relation to other Yourdan notations. In the figure, the *cross-notational links* `withType` and `withEvent` originate from concepts in DFD notation; it is also possible, however, to have cross-notational links start from concepts in other notations and end in concepts in DFD notation (as is the case in the figure for the link `respondedBy` of `ExternalEventType`). The

gel

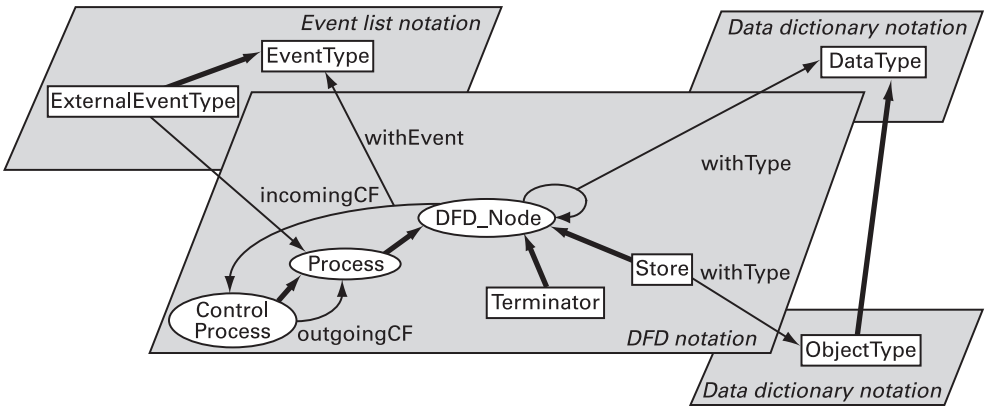


Figure 3.6
The DFD notation cross-related to other Yourdan notations

Typo: The label of the area behind "ObjectType" in Figure 3.6 should be "ERD notation"

method engineer completing the diagram has to decide to which notation each cross-notational link in the diagram belongs. Here, we assume that they belong to the DFD notation. The reader may have noticed that a cross-notational link is not rendered any differently than other notational links (e.g., the `dataflow` link of the DFD notation). The decision to put certain node and link types into a certain notation is made to manage the complexity of a modeling notation within a method. It would also be possible to define a single "Yourdan" notation that encompassed all node and link types (i.e., the union of DFD, ERD, and all other Yourdan notations). Such a unified notation, however, would be too complex for human modelers to handle. Experience has shown that modeling is more successful when the modeler concentrates on the specific aspects of the modeling task rather than the global task of modeling a complex system.

DFD notation is interrelated with other notations in various ways. First, data stores have *object types* as data structures. The class `ObjectType` is a reference to ERD notation, where it serves as the common superclass of `EntityType` and `RelationshipType`. The data types attached to data flows can also be regarded as cross-notational links: Data types are defined in *data dictionaries*.[12] Second, the event types attached to incoming control flows of DFD control processes can be external event types enumerated in the event list.

Model level for an example DFD and event list

source

```
MyDFD in PreliminaryBehavioralModel with
  containsDFDNode
    n1: CUSTOMER;
    n2: CREDITCARDCOMPANY;
    n3: UpdateAccounts;
    n4: Accounts
end

CUSTOMER in Terminator with
  dataflow
    d1: UpdateAccounts
end

CREDITCARDCOMPANY in Terminator with
  dataflow
    d2: UpdateAccounts
end

UpdateAccounts in Process with
  dataflow
```

```
    d3: Accounts
end

Accounts in Store end

CUSTOMER!d1 with
  withType
    datatype: payment
end

CREDITCARDCOMPANY!d2 with
  withType
    datatype: maximumcredit
end

UpdateAccounts!d3 with
  withType
    datatype: verifiedpayment
end

payment in DataType end
maximumcredit in DataType end
verifiedpayment in DataType end

MyEventList in EventList with
  containsEvent
    ev1: E1;
    ev2: E2
end

E1 in FlowEventType with
  agent a1: CUSTOMER
  eventtext t: "A customer makes a payment"
end

E2 in TemporalEventType with
  eventtext t: "At 18:00 the invoices of current orders are sent
  out"
end
```

slides

3.15.4 Modeling Intranotational Constraints

An *intranotational constraint* is a constraint ranging over concepts of a single notation (e.g., the ERD notation). In this section, I consider only constraints that restrict

the set of allowed diagrams.[13] Some of these constraints are already imposed by the choice of Telos as the language. For example, the ERD notation defines

```
RelationshipType in Node isA ObjectType with
  connectedTo
    role: EntityType
end
```

Telos axiom 3.14 then requires that any instance of `RelationshipType` that uses the `role` link must end in an instance of `EntityType`. Furthermore, any instance of `RelationshipType` is also an instance of `ObjectType` (by axiom 3.13).

Beyond such predefined rules of well-formedness, a notation definition should also include internal constraints. With ConceptBase, there are two principal strategies for encoding notational constraints. First, they can be represented as ordinary class constraints. Second, they can be encoded as query classes, which then return the "violators" of the constraint. To demonstrate this principle, let us consider the following constraint for the ERD notation:

C1. Any entity type must have at least one describing attribute.

If we want to realize this condition as a class constraint, we have to instantiate the concept `EntityType` as an instance of the built-in object `Class`, which defines the attribute category `constraint`:

```
EntityType in Node,Class isA ObjectType with
  connectedTo
    ent_attr: Domain
  constraint
    c1: $ forall e/EntityType
        exists a/EntityType!ent_attr From(a,e) $
end
```

The foregoing class constraint is a realization of constraint C1, but it has one important drawback: No database violating the constraint will be accepted by the ConceptBase system. In practical modeling applications, however, we want violations to be tolerated *during* the modeling process, that is, when we are constructing the diagrams. During the modeling process, the models are typically incomplete. Only at certain milestones is the incompleteness expected to be resolved, and only at those points are violations to be regarded as unacceptable. Strict enforcement of the constraint just defined would prevent us from defining any entity type without providing at least one entity attribute. One solution to the problem is to provide the constraint to the system only at the appropriate milestone in the modeling process. If the

constraint is fulfilled at that point, it will be accepted by the system; otherwise, an error message will be generated.

As noted previously, there is a second alternative for solving the problem, and it is in most cases preferable: Represent the constraint as a query class, which then returns the violators. This method works for all constraints that have the format

```
forall X A(X) ==> B(X)
```

The violators of such constraints are all `X` that fulfill the condition

```
A(X) and not B(X)
```

In the foregoing example, we can apply the pattern as follows for `X=e`:

`A(e)` = `In(e,EntityType)`, where the `In` predicate is implicit by the typed quantification `e/EntityType`
`B(e) = exists a/EntityType!ent_attr From(a,e)`

Given this transformation, we can mechanically create the query class, which computes the violators of the constraint:

```
EntityTypeWithoutAttribute in QueryClass isA EntityType with
  constraint
    c1: $ not exists a/EntityType!ent_attr From(a,~this) $
end
```

The answer variable `~this` assumes the role of the variable `e` in the first solution presented, as elaborated in the section on query classes. The query class representation eliminates the problem with the first solution, in that at any point in time, a modeler can ask for the violators of the original constraint. It is a matter for the process model of the method to define when the query class has to return an empty answer.

Being more complex than ERD notation in terms of number of concepts, the DFD notation also features more intranotational constraints. Let us consider the following constraints:

C2. Any process must have at least one ingoing and one outgoing dataflow.

C3. Data flows between terminators are forbidden.

C4. Data flows must have data types attached to them, with the exception of data flows starting from or ending in data stores.

The following query classes encode these constraints. (Constraint C2 is split into two query classes for the sake of readability and usability.)

```
ProcessWithoutInput in QueryClass isA Process with
  constraint
    c21: $ not exists d/DFD_Node!dataflow To(d,~this) $
end

ProcessWithoutOutput in QueryClass isA Process with
  constraint
    c22: $ not exists d/DFD_Node!dataflow From(d,~this) $
end

TerminatorWithForbiddenCommunication in QueryClass isA Terminator
with
  constraint
    c3: $ exists t/Terminator d/DFD_Node!dataflow From(d,~this)
        and To(d,t) $
end

DataflowWithoutLabel in QueryClass isA DFD_Node!dataflow with
  constraint
    c4: $ not (exists s/Store From(~this,s) or To(~this,s)) or not
        (exists dt/DataType (d withType dt)) $
end
```

The level of complexity of a notation can be measured in terms of the number and size of the intranotational constraints it imposes. The more such constraints are defined, the more is known about syntactically correct models in that notation. As a consequence, there are also more opportunities to violate the constraints, which makes the notation more difficult to handle. The method engineer should take such considerations into account when defining a notation.

3.15.5 Modeling Internotational Constraints

In contrast to the intranotational constraints just discussed, which involve concepts from only a single notation, *internotational constraints* refer to concepts from more than one notation. Since Telos provides a uniform framework for all notations (as well as all models, data, and process executions), such constraints look like intranotational constraints. In fact, the decision to assign concepts to more than one notation is arbitrary and made for the sake of readability of the models and understandability of the notations themselves.

A simple example of an internotational constraint is the following:

C5. Each data store must have an object type associated with it and vice versa.

Two query classes realize this constraint:

```
ObjectTypeWithoutStore in QueryClass isA ObjectType with
  constraint
    c21: $ not exists s/Store (s withType ~this) $
end

StoreWithoutType in QueryClass isA Store with
  constraint
    c21: $ not exists o/ObjectType (~this withType o) $
end
```

Realized in this way, the constraint uses the cross-notational link `withType` that has been defined as part of the DFD notation. The concept `ObjectType` is part of the ERD notation, whereas `Store` is a concept in the DFD notation. There is nothing noteworthy or exceptional in regard to the intranotational constraint.

The concepts `ObjectType` and `Store` are closely related. They have to occur in pairs. So apparently, they are two aspects of the same abstract thing. This observation leads indeed to a principle of method engineering that can be represented in the notation definition level and then exploited for the detection of internotational constraints.

In our example, both `ObjectType` and `Store` are incarnations of an abstract data concept. Object types refer to the interrelationships among data concepts (as handled by the ERD notation). Stores have to do with the location of data in the network of communication processes (as handled by the DFD notation). The common abstract object is the manifestation of the artifact focus mentioned earlier: The models and notations are interrelated because they make statements about the same artifact (e.g., an information system). Hence, it is logical to enrich the notation definition level by adding such abstract concepts and then investigate the links between their incarnations in different notations. The following richer notation definition level may be suitable for system analysis methods like Yourdan's.

Notation definition level with abstract concepts

```
Node in Individual with
  attribute
    connectedTo: Node
end

Project in Individual with
  attribute
    produces: Model
end
```

source *

gel

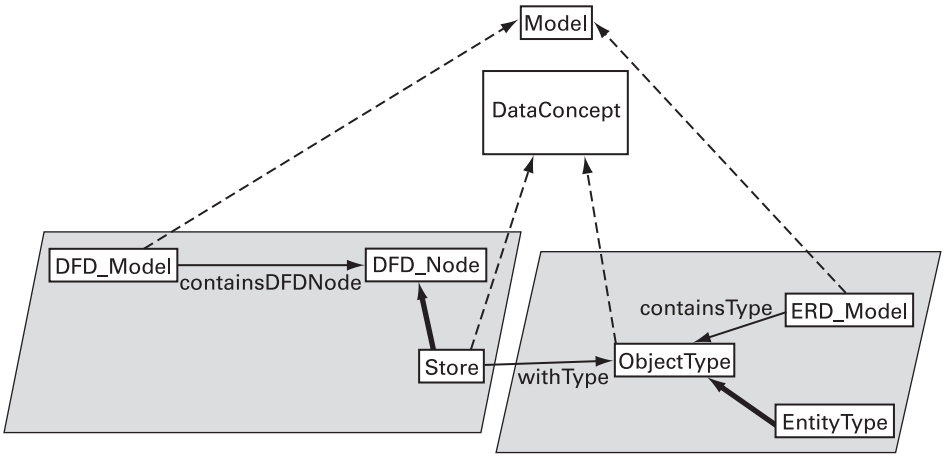


Figure 3.7
Incarnation of `DataConcept` in DFD and ERD

```
Model in Individual with
  attribute
    contains: Node
end

DataConcept in Individual isA Node end
ActivityConcept in Individual isA Node end
ControlConcept in Individual isA Node end
```

slides

The three abstract concepts here are data concept, activity concept, and control concept. Whenever two different notations instantiate the same abstract concept, it becomes necessary to check whether a cross-notational link and internotational constraints are required. In the case of `ObjectType` and `Store`, we have the scenario displayed in figure 3.7.

If we were dealing with the data concept in a single notation, there would be no need to distinguish the `Store` aspect of data from the `ObjectType` aspect. Because we are separating them, however, we need to synchronize their definitions. In Telos, the synchronization is performed by the cross-notational link plus the internotational constraints, as shown previously. The Yourdan method includes more examples of the phenomenon. For example:

- Process and process specification are both incarnations of the activity concept.
- Control process and state transition diagrams are both incarnations of the control concept.

*) The Telos source at the link for figure 3.7 is slightly different and more general than the source sample in the text.

The focus on artifacts in method engineering has another consequence: When two models involve the same artifact (like an information system), they need to be synchronized. However, assume that we are maintaining models of different modeling projects in the same repository. How can we keep them separate from one another? The answer lies in the objects `Project` and `Model` of the notation definition level. If the repository stores details of several independent projects, then the notational constraints of each project need to be made "project-aware." For example, if a modeling project involves both DFD and ERD models and contains a DFD model that itself contains a store, then there must be an object type linked to this store that is defined in an ERD model of the same modeling project:

```
YourdanProject in Project end
StoreWithoutTypeV2 in QueryClass isA Store with
  constraint
    c21: $ exists p/YourdanProject dfd/DFD_Model (p produces dfd)
         and(dfd containsDFDNode ~this) and not (exists erd/
         ERD_Model o/ObjectType (p produces erd) and (erd
         containsType o) and (~this withType o) $
end
```

The object `YourdanProject` is at the notation level. It subsumes all modeling projects using Yourdan notations.

Project-aware intranotational constraints such as the one just defined can be written in various combinations and versions. For example, assume that a method engineer wants to check whether a given Yourdan project violates constraint C2. In this case, a generic query class is a facility for formalization of the constraint:

source

```
DFDwithUntypedStore in GenericQueryClass isA DFD_Model with
  parameter,computed_attribute
    project: YourdanProject
  computed_attribute
    store: Store
  constraint
    c21: $ (~project produces ~this) and (~this containsDFDNode
         ~store) and not (exists erd/ERD_Model o/ObjectType
         (~project produces erd) and (erd containsType o) and
         (~store withType o) $
end
```

The query class will return the names of all DFD models in the Yourdan project, in addition to the name of the project to which they belong and the name of the

gel

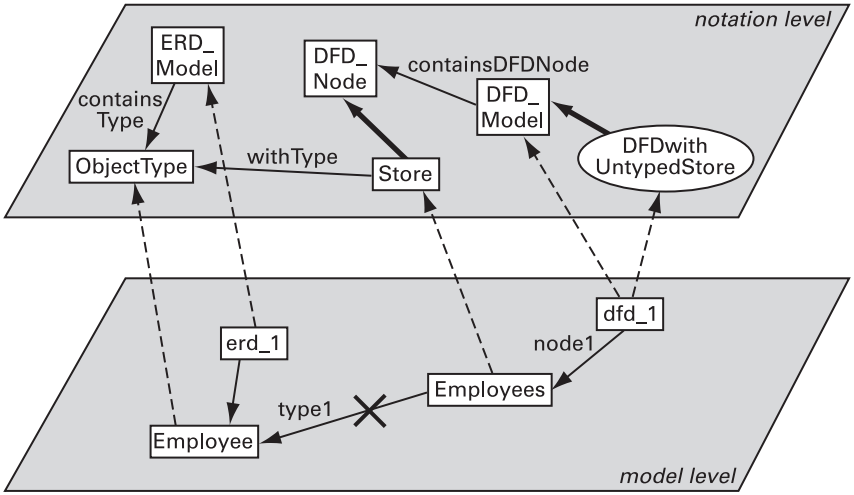


Figure 3.8
Internotational constraint at notation level

violating store. The parameter `project` can also be left void, that is, unused in the query call. In that case, the query will scan all known Yourdan projects in the repository. The attribute category `computed_attribute` ensures that the name of the project and data store involved are returned together with the name of the violating DFD model.

3.15.6 Multilevel Statements to Express Semantics

The queries discussed in the foregoing constrain the set of models that are acceptable in an information systems development method. A violation of some intra- or internotational constraint can be detected just by analyzing the models. We can regard the set of all notational constraints on a set of modeling notations as the *syntax* of the modeling notations. A characteristic of the query classes for notational constraints is that their variables range over objects at the IRDS model level.

Figure 3.8 displays a representation of the query class `DFDwithUntypedStore` of section 3.15.5, an example of an internotational constraint. The query is defined at the notation level: It is a subclass of `DFD_Model` and refers to objects defined at the notation level. Its variables (like `~this` and `store`) range over objects at the model level. The query returns those DFD models that violate a certain internotational constraint, namely, that all data stores of a particular DFD must be linked to an object type in an ERD model belonging to the same system development project. All intra- and internotational constraints share the property of being *defined at the notation*

level and having variables ranging over objects at the model level. However, there are more ways to express knowledge about modeling notations, namely, to express *semantics* of certain notational symbols.

In predicate logic, the semantics of a logical theory, that is, a collection of logical formulas, are based on sets of objects. In particular, mathematical relations with matching arity interpret predicates. In the restricted framework of Datalog, predicates are interpreted by sets of facts in which variables in the predicates have been replaced by constants. Consider the example predicate `In(x,Employee)`. A possible interpretation *ext* computed by the fixpoint algorithm for Datalog could be:

```
ext(In(x,Employee)) = {In(bill,Employee), In(mary,Employee)}
```

Remembering that `Employee` is the name of a class, we can say that the interpretation of the class `Employee` is the set `{bill,mary}`. Analogously, attribute predicates can be interpreted by relational facts:

```
ext(A(e,salary,s)) = {A(bill,salary,1000),A(mary,salary,2000)}
```

We can easily extend this type of semantics to the other IRDS abstraction levels; for example, the semantics of the predicate `In(e,EntityType)` are computed using the Datalog fixpoint mechanism and yield sets like

```
ext(In(e,EntityType)) = {In(Employee,EntityType),
                         In(Project,EntityType}
```

There are, however, properties of notations that do not fit directly into this simple kind of semantics. For example, the cardinality of role links in ERD notation is checked at the data level, whereas the semantics of cardinality tags like 1:*n* ought to be defined at the notation level.

A solution to this problem is the use of metalevel formulas, which are formulated at a metaclass level and can have variables ranging over objects that are instances of instances of the objects occurring in the formula. Figure 3.9 shows a relationship type `worksFor` whose role link to `Employee` has a 1:*n* cardinality. The example in the data level violates the cardinality. Hence, the semantics are dependent on the database content, whereas we consider cardinalities to be part of the notation. The following metalevel constraint defines the semantics:

source

```
Cardinality in Class with
  constraint
    c1: $ forall R1,R2/RelationshipType!role
                  RT/RelationshipType ET1,ET2/EntityType
                  x/VAR
                  From(R1,RT) and To(R1,ET1) and
```

gel

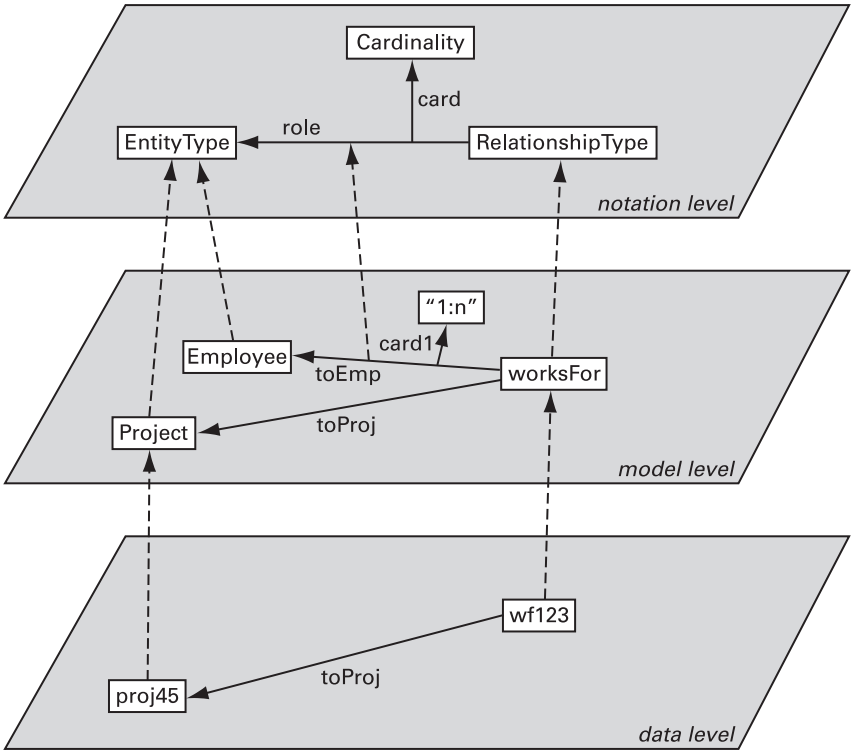


Figure 3.9
The semantics of the notational concept "cardinality"

```
                    From(R2,RT) and To(R2,ET2) and
                    not (R1 == R2) and
                    (R1 card "1..n") and (x in ET2)
            ==>
              ( exists y/VAR r1,r2/VAR rt/VAR
                  (r1 in R1) and (r2 in R2) and
                  (rt in RT) and (y in ET1) and
                  From(r1,rt) and To(r1,y) and
                  From(r2,rt) and To(r2,x)
              )
        $
end
```

The constraint refers to the notation-level concepts `EntityType`, `Relationship-Type`, `card`, and `"1:n"`. All other objects are ranged over by variables. The variables

R1, R2, RT, ET1, and ET2 range over objects at the model level. Possible values for ET1 and ET2 are Employee and Project, respectively. A possible value for RT is worksFor. The variables x, y, r1, r2, and rt range over objects at the data level. For example, a possible value for x is proj45. So whenever an object like proj45 (x) is defined to be an instance of Project (ET2), then there must be at least one instance y of class Employee (ET1) that stands in the worksFor (RT) relationship to it. The cardinalities 1:1, 0:1, etc. can be expressed in a similar way. ConceptBase will partially evaluate metalevel formulas to compile them into a set of simple formulas ranging over just one IRDS level, as explained in section 3.13.

Another example of the use of metalevel formulas is the definition of disjoint specialization in UML class diagrams (see section 1.3 and figure 1.5). UML has a variant of class specialization that requires any two subclasses of the same superclass to be disjoint; that is, no object x may be an instance of two such classes at the same time. Whereas specialization in UML is defined at the notation level, the object x is part of the data level. The Telos realization of the semantics of disjoint specialization is included on the CD-ROM that accompanies the volume.

Besides specifying the semantics of notational link symbols, metalevel formulas create opportunities to clarify the semantics of node symbols as well. Consider again the class EntityType defined at the notation level. Its Herbrand interpretation *ext*(EntityType) is the set of all its instances. Instances of instances of EntityType are called *entities* and are located at the data level. Let us use the predicate (x [in] mc) to express that the object x is an instance of some instance of mc. Then the concept of an entity is formalized as

```
forall x/Individual c/EntityType
(x in c) ==> (x [in] EntityType)
```

Analogously, the set of all *values* is the set of all objects that are instances of some instance of the class Domain:

```
forall x/Individual c/Domain
(x in c) ==> (x [in] Domain)
```

Both formulas are structurally derived from the generic formula

```
forall x/Individual c/Individual mc/Individual
(x in c) and (c in mc) ==> (x [in] mc)
```

By coding the generic formula in ConceptBase, one can query the data level independent of the model level. In the ERD case, one can ask for all entities or all values that are currently known *regardless to which entity type or domain type they belong*. What's more, one can ask for all entities that are related to some other entity regard-

source

gel

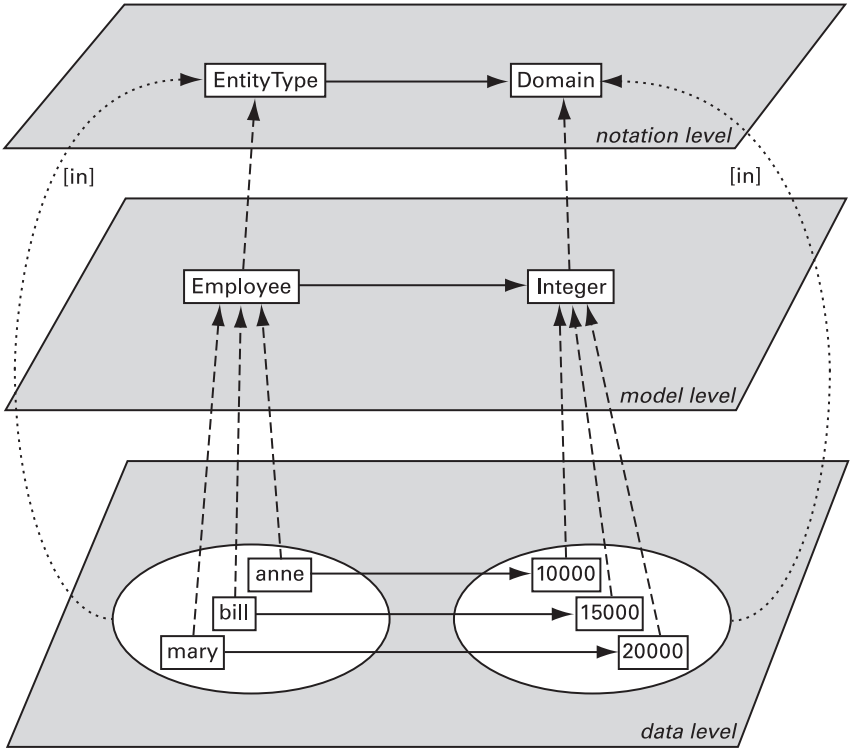


Figure 3.10
Understanding the data level from the notation level

less of the underlying ERD model (i.e., database schema). An example is shown in figure 3.10: The currently known entities are `mary`, `bill`, and `anne`. The values (i.e., instances of instances of `Domain`), are `10000`, `15000`, and `20000`. A ConceptBase implementation of the predicate `(x [in] mc)` can be found on the accompanying CD-ROM.

Note that not all metalevel formulas are eligible for partial evaluation. The case of ERD shows that the semantics of certain notational symbols can be expressed by constraints in Telos. Moreover, in ERDs, concepts at the notation level can be used to query objects at the data level directly. The semantics of dynamic models like data flow diagrams are more difficult to capture than that of static diagrams such as ERDs, because they express complex transformations. Mapping models of such dynamic notations to simulation environments appears to be more conducive to capturing their semantics.

3.15.7 Modeling Process Model Notations

slides

The notations discussed so far are languages for recording models about some artifact. The models are restricted by constraints, some of them ranging over models recorded in different notations. In addition to internotational constraints, there is another relevant connection between models: Some models are created out of others. In the case of system development, the code of a program module is constructed from the chart of some module and the specification of the processes that are part of the module. There is some *process model* that tells the modeler from which input models a new output model can be created. We regard such a process model to be part of the engineering of a method.

Process models can be *descriptive* (a structured documentation of what steps have been performed during a modeling project) or *prescriptive* (a set of rules that define which development steps are allowed in a given situation). They play a major role in any modeling method because they are the basis for refining the modeling method as a whole and for defining new services like the traceability of model details.

A common feature of process models is the blurring of abstraction levels. Consider the following excerpt from a fictional software development project:

10-Oct-2004, 10:23: Mary has completed interviews with the user group and publishes her interview report REQ14.doc.

12-Oct-2004, 15:56: John has read the report REQ14.doc and produces the event list EL1 from it.

13-Oct-2004, 9:45: John produces the preliminary behavioral model MyDFD1 from EL1.

The statements themselves are concrete (i.e., data level in the IRDS framework) and cannot be instantiated. They constitute the *trace* of an actual modeling project. On the other hand, some parts in the trace refer to concepts that we previously classified into the model level: the report REQ14.doc, the event list EL1, and the preliminary behavioral model MyDFD1 are all models whose content is at class level (i.e., they do have instantiations). The reason for this mixture of levels is that the very activity we are observing is producing models, not concrete data.

The concrete statements of a process trace follow patterns. The patterns for the above example could be:

An interviewer completes interviews and publishes interview reports.

An analyst reads interview reports and produces event lists.

An analyst produces preliminary behavioral models from event lists.

A collection of such statements (for modeling projects) is called a process model. We can continue this abstraction and conclude that all three statements of the process model are examples of the generic statement

An expert reads models and creates models.

which defines how process models should look; hence it is part of a *notation for process models*.

Note that the term "model" occurs in the notation definition level of the modeling products. Hence, the mixture of levels that was observed at the trace continues in the more abstract statements. As such, the statements about model production have the same nature as the statements that were used to describe various abstractions on the product side (data, models, notations, notation definition levels). They do have one extra property: They relate in a new way to objects defined at the product side. Figure 3.11 relates production-oriented process models (right side of figure) to their products (left side). The IRDS abstraction levels are applicable to the process models as well to the products, but the levels are *skewed* by one level on the two sides of the

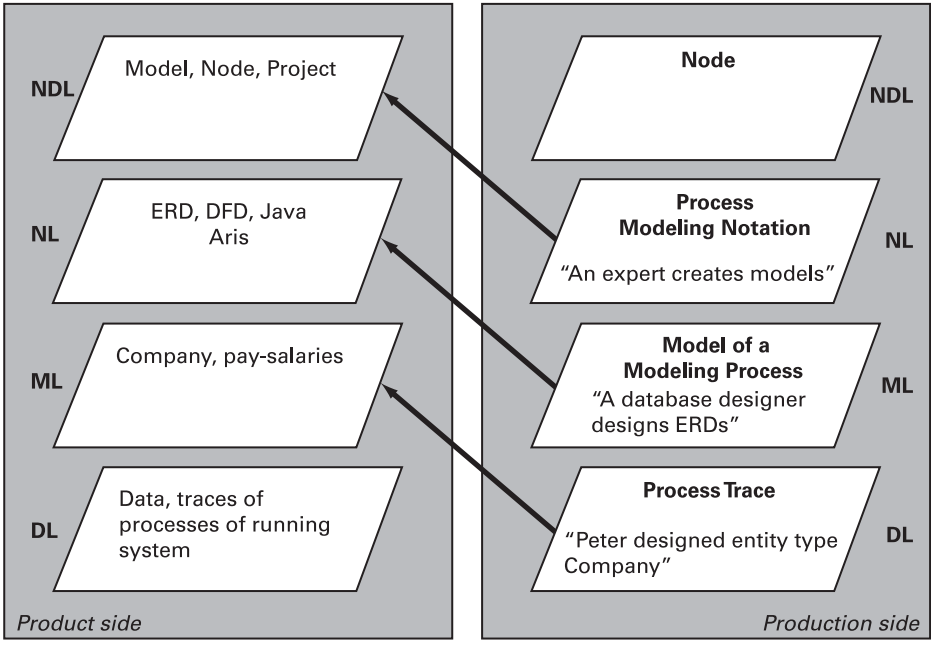


Figure 3.11
Production and product side of modeling. *Note*: NDL: notation description level; NL: notation level; ML: model level; DL: data level

figure. The reader should note the similarity of the situation here to that surrounding the fact `(EntityType attribute/author PeterChen)`, discussed in section 3.3. There, as in this case, objects from different IRDS levels are associated with one another.

The origin of the skewing of levels in the figure lies in the nature of models. A model can, on the one hand, be seen as a collection of classes whose instances are possible interpretations of the classes. On the other hand, the objects in a model are the input to and output of various development steps; that is, they are data themselves.

sources

Folder SPM on the companion CD-ROM to the volume contains an example relating to software process modeling, that is, a notation plus examples on how to specify the modeling steps of a notation. The example also features an almost complete representation of the notations of the Yourdan method. The reader is advised to take the definitions as an example: They show how the process model integrates with the notations defined earlier. The notational constraints are realized as query classes and can be attached to individual modeling steps: Whenever such a step is executed, the specified constraints should be checked.

Simplified notation level for a process model

source

```
ModelingStep in Node with
  connectedTo
    precedes: ModelingStep
  attribute
    inputObject: Node;
    outputObject: Node;
    postcondition: QueryClass
end

Methodology with
  attribute
    containsStep: ModelingStep
end

Project with
  attribute
    produces: Model;
    uses: Methodology
end
```

Model level for a process model (excerpt)

```
YourdanProject in Project with
```

```
  produces
    eventlist: EventList;
    prelimbehavioralmodel: PreliminaryBehavioralModel;
    processspecification: ProcessSpecification;
    erd: ERD;
    std: STD;
    sourceProgram: SourceProgram;
    testPlan: TestPlan;
    testData: TestData;
    testResult: TestResult
  uses
    methodology: YourdanMethod
end

YourdanMethod in Methodology with
  containsStep
    extractEventList: ExtractEventList;
    mapToDFD: MapToDFD;
    specifyProcess: SpecifyProcess;
    codeProcess: CodeProcess;
    createTestPlan: CreateTestPlan;
    executeTest: ExecuteTest
end

YourdanStep in ModelingStep with
  attribute
    input: Proposition;
    output: Proposition;
    starttime: Real;
    endtime: Real
end

ExtractEventList in ModelingStep isA YourdanStep with
  inputObject
    input1: UserRequirements
  outputObject
    result1: EventList
end

ProduceLeveledDFD in ModelingStep isA YourdanStep with
  inputObject
    prelimDFD: PreliminaryBehavioralModel
```

```
  outputObject
    levDFD: LeveledDFD
  postcondition
    r1: NotLeveledNotSpecifiedProcess;
    r2: BothLeveledAndSpecified;
    r3: DanglingInputNotMatchedAtProcess;
    r4: DanglingOutputNotMatchedAtProcess;
    r5: InputNotMatchedByDangling;
    r6: OutputNotMatchedByDangling;
    r7: IllegallyLeveledProcess
end
```

Data/trace level for a process model (excerpt)

```
step1 in ExtractEventList with
  result1 el: MyEvent_456
  starttime t1: 3.0
  endtime t4: 4.0
end

step2 in MapToDFD with
  input1 el: MyEvent_456
  output1 dfd: MyDFD_023
  starttime t1: 5.0
  endtime t4: 7.0
end
```

3.15.8 Managing Modeling Processes Using Metrics

slides

The uniform representation of the development trace, the software process model (here, the Yourdan example), and the process model notation together with the product counterparts (example models, modeling notations, notation definition level) allows complex modeling situation to be analyzed by means of queries. In particular, one can express metrics for any of the objects defined in a project's repository. As example, consider the metric `CodingProductivity`, which measures the number of lines of code created per time unit in a coding step. The following query class realizes the metric in terms of two other metrics, `Duration` and `ProgSize`:

source

```
ProgSize in GenericQueryClass isA Integer with
  parameter,computed_attribute
    whatProg: SourceProgram
  constraint
```

```
    c: $ (~this in COUNT_Attribute[~whatProg/objname,
       SourceProgram!lines/attrcat]) $
end

Duration in GenericQueryClass isA Real with
  parameter,computed_attribute
    step : YourdanStep
  constraint
    c1 : $ exists t1,t2/Real (~step starttime t1) and (~step
         endtime t2) and (~this in MINUS[t2/r1,t1/r2]) $
end

CodingProductivity in GenericQueryClass isA Real with
  parameter,computed_attribute
    cstep: CodeProcess
  constraint
    c1: $ exists sp/SourceProgram dur/Real size/Integer (~cstep
        program sp) and (dur in Duration[~cstep/step]) and (size
        in ProgSize[sp/whatProg]) and (~this in DIV[size/r1,dur/
        r2]) $
end
```

The queries in this query class exploit the built-in queries (Jarke, Jeusfeld, and Quix 2003) of ConceptBase for simple arithmetic (PLUS, MINUS, DIV). The attribute categories `parameter` and `computed_attribute` in combination ensure that the query answer can be interpreted even if the parameter is left undefined at query call time. For example, the query class `CodingProductivity` (without a filler for parameter `cstep`) will return the coding productivities of all coding processes (attached as computed attribute `cstep`).

Metric queries can be integrated into process models as postconditions of modeling steps very much like the queries for checking notational constraints. More insights on the use of metrics in software development are offered in Fenton and Pfleeger 1999.

Chapter 8 discusses in more detail how metrics can be incorporated into design processes supported by the ConceptBase repository. The application examined in that chapter is a repository for managing the quality of data warehouse systems.

3.16 Discussion and Conclusions

This chapter presented the use of the ConceptBase repository for metamodeling and the engineering of modeling methods. The approach as presented here is focused on

notation definition, and one might ask whether this is sufficient to cover all aspects of a complete method like Yourdan's Modern Structured Analysis. The syntactic features of the Yourdan notations for entity-relationship diagrams, data flow diagrams, process specifications, state transitions diagrams, and so on can be translated rather easily into suitable Telos metaclasses. Correctness rules (called balancing rules by Yourdan) map straightforwardly into query classes. There are, however, a few soft rules in Yourdan's method that are naming conventions to make models easier to read for humans (e.g., the rule that the name of a process should be a verbal phrase) and that do not transfer readily to Telos. Such rules are vague by nature of human communication, and a violation of such soft rules is acceptable to a certain degree.

Another aspect of method engineering besides notation definition is instructional examples. Textbooks on modeling methods are cluttered with examples that demonstrate features of the methods. The presentation of such examples is in fact easily supported by the repository approach, since example models are just one IRDS level below the notations related to those models and are represented in the same Telos framework as the notations themselves. They have an even bigger role with our approach, as the relation of example models to notations is now fully formalized as a Telos instantiation relation. During the design of a new notation, the creation of example models validates the correctness and usefulness of certain notational constructs. Hence, not only the student of a method is supported, but also the method engineer in her search for the right notation definition.

A method description is incomplete if it does not contain some kind of step-by-step instructions on how to proceed with a modeling project. Process models, as presented in the chapter, can fill this need. The integration of correctness rules as postconditions to modeling steps goes beyond the original description of the Yourdan method. Originally, the rules were formulated as part of the notations associated with the models. With process models, one can precisely design when certain checks are to be performed on the models created so far.

Finally, the quality of a collection of models can be managed using metric definitions. This feature of ConceptBase covers some of the soft aspects of model development (in teams). In the case of the Yourdan method, there are rules regarding the desirable level of complexity in diagrams that easily map into metric definitions. A similar metric technique can be applied to controlling the quality of modeling steps (e.g., by measuring their productivity).

So what remains uncovered as we finish the chapter? Insights on proper usage of modeling techniques are hard to represent as part of a notation. Proper usage depends on the modeling expert's knowledge of the application domain and cognitive skills. Modeling textbooks cover proper usage in case studies with extensive comments in which reasons for certain design decisions are elaborated upon. Proper usage of modeling methods alone does not ensure good usage. One has to introduce

experience knowledge of successful usages. One way to support experience knowledge with the repository approach is to keep the results of previous and current modeling projects within the same repository. Novice modelers can then search the archive of old projects for solutions to certain problems that may come up in current ones on which they are working. Of course, the models of old projects would need to be annotated and indexed for this type of use. Moreover, design decisions should be made an integral part of the process models, and documentation of such decisions should include the rationale behind them.

A frequently cited aspect of modeling methods is their *pragmatics*, that is, what resources are needed to apply them, and in particular, how much time should be spent on teaching the methods to users. If the resource requirements for individual modeling steps are known, then they can be included in the method definition. Essentially, one has to create a plan for project management in which resources, time, and cost play an important role. The plan can be encoded using a Telos-based notation for workflow management. However, such a notation doesn't really solve the problem of how to allocate scarce resources (e.g., installing a help desk for questions on a modeling method).

Finally, the decision to use Datalog as a foundation for the semantics of the Telos definitions deserves a critical discussion. The perfect-model semantics of Datalog eliminate any formal ambiguity in interpretation (via stratification) and are always finite. The latter restriction precludes a complete coverage of the semantics of dynamic notations such as state transition diagrams or program code. Still, the syntactic features of such dynamic notations and their interrelationships with other notations can be well represented. A semantics specification powerful enough for dynamic-modeling notations would stand in conflict with efficiency of model management.

Perfect-model semantics essentially allow queries to be answered, that is determining which objects of a given finite extension fulfill the membership condition of the query. They do not support reasoning about elements of a model (e.g., whether some class definition is subsumed by another, as in section 1.3). One can argue that such reasoning services can be attached to the model repository as external tools.

Notes

1. Telos statements can be expressed as binary predicate facts in infix form. Basically, the statement number is omitted because it is system-generated and carries no meaning as such. Later in the chapter, we show how to map predicate facts into a reified form, called P-facts. This reified form is crucial for defining the semantics of a set of Telos statements and for providing facilities for engineering the semantics of modeling notations.

2. The metaclass `DomainOrObjectType` is a superclass of both `ObjectType` and `Domain`. Its definition is presented in section 3.4, on Telos frame syntax.

3. The latter object is a predefined Telos object with name instanceOf, whose definition can be found in section 3.8. There are only five predefined Telos objects.

4. The advantage of using explicit quantifiers is that one can then also formulate logical expressions with nested quantifications. It can be shown that any such formulation can be transformed into a set of Datalog-style deductive rules that have the same semantics.

5. We say that constants are interpreted by themselves to express that we do not interpret them by some (real world) object as in classical logic. Instead, the meaning of a constant such as "10" is just "10."

6. There are, however, nonstratifiable rule sets that are not at all paradoxical. They even have a unique model. I refer the reader to articles describing stable model semantics for more information.

7. The constraint text itself is an instance of `MSFOLconstraint`. We omit the Telos frame that defines this instantiation. ConceptBase will include the instantiation automatically. The same holds for rule texts, which are automatically classified into the class `MSFOLrule`.

8. The query class with the explicit variable `~this` is shown for illustration only. It does not represent a syntactically correct query class. Note that the variable `~this` ranges over all instances of `Department`, that is, the superclass of which the query class is a subclass.

9. The predicate (`p in Proposition!single`) is translated from the variable range `p/Proposition!single`.

10. Full-fledged active database systems provide more event types and a complex event composition language to express sequences of events (`e1` after `e2`), logical connectives (`e1` or `e2`), and temporal events. More on active databases can be found in Paton 1999.

11. *Transaction* is a technical term in the database domain. It stands for a sequence of operations on the database that is executed as a whole.

12. I do not discuss the data dictionary notation in great detail here.

13. One can regard such constraints as part of the syntax definition for a notation. The syntax for a given notation circumscribes the set of allowed statements (= models) in that notation.

References

Ceri, S., G. Gottlob, and L. Tanca. 1990. *Logic Programming and Databases*. Heidelberg: Springer-Verlag.

Chen, W., and D. S. Warren. 1996. "Computation of Stable Models and its Integration with Logical Query Processing." IEEE Transactions on Knowledge and Data Engineering 8, no. 5: 742–757.

Fenton, N. E., and S. L. Pfleeger. 1999. *Software Metrics—A Rigorous and Practical Approach*. 2nd ed. Boston: PWS.

Greenspan, S. 1984. "Requirements Modeling: The Use of Knowledge Representation Techniques for Requirements Specifications." Ph.D. diss., University of Toronto.

Jarke, M., M. Jeusfeld, and C. Quix, eds. 2003. *ConceptBase V6.1 User Manual*. Available at ⟨http://www-i5.informatik.rwth-aachen.de/CBdoc/userManual/⟩.

Jeusfeld, M. 1992. *Änderungskontrolle in deduktiven Objektbanken*. St. Augustine: Infix-Verlag.

Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis. 1990. "Telos—A Language for Representing Knowledge about Information Systems." *ACM Transactions on Information Systems* 8, no. 4: 325–362.

Paton, N. W. 1999. *Active Rules in Database Systems*. New York: Springer-Verlag.

Yourdan, E. 1989. *Modern Structured Analysis*. Englewood Cliffs, NJ: Prentice Hall.

4 Conceptual Modeling in Telecommunications Service Design

Armin Eberlein

This chapter demonstrates how conceptual models in Telos can be used to create an intelligent tool called RATS that helps during the development of new telecommunications services. The focus of RATS is on the requirements engineering phase which has provided the most challenges in the past. The RATS tool can be integrated with currently available development approaches including formal methods. The chapter describes the overall architecture of the tool and then focuses on the Telos conceptual models that have been created. Emphasis is placed on the approach taken to providing intelligent support to the telecommunications service developer.

4.1 Introduction

Telecommunications technologies have brought about tremendous changes during the last 150 years and are still developing rapidly (Bellamy 1991). These technologies have influenced culture and social life by offering new means of interaction and communication among users distributed around the world. Since its invention in 1876 by Alexander Graham Bell, the telephone has experienced profound changes. Over the years, the telecommunications network has covered most of the globe. Recent advances in telephone mobility and quality over longer distances have resulted in a highly complex communication medium that everybody takes for granted nowadays.

As new telecommunications technologies become available, additional services and service features are possible. However, the implementation of such features has persistently proved difficult and has become one of the major challenges for telecommunications service providers (SCORE 1995). As a result, many service providers offer their customers what is easy to implement, rather than what is possible and what customers want. The introduction of any new service causes fears about how the service will "behave" on the network, that is, how it will interact with already existing services (Bouma and Velthuijsen 1994). Will the new and existing services

coexist in peace, or will they have major unwanted interactions, possibly bringing down the whole network? The situation is further complicated by the heterogeneity of telecommunications networks. Services have to handle equipment of different functionalities and age, produced by a variety of vendors in a worldwide distributed network (Reed, De Man, and B. Møller-Pedersen 1989). Furthermore, there are always fears about the consequences of bugs in the service implementation. In January 1990, a single misplaced statement in an AT&T switching system caused the entire Eastern seaboard of the United States to lose its telephone network for several hours (Neumann 1995).

However, telecommunications is not the only industry that struggles with such problems; so do other industries that use software (Standish Group 1994). The fact of the matter is that telecommunications systems and services are founded on large, very complex, and ever-growing software. The software component of these systems constitutes at least 70 percent of the total effort required to design them (Reed et al. 1992). In other words, many of the problems in telecommunications are actually inherited from software engineering.

Most people who have been seriously engaged in the study and development of software systems have concluded that one of the most problematic tasks in their development is understanding the requirements of the system being developed and correctly transforming them into code (Sommerville 2000). Not only the functionality of the system itself, but also the environment in which it must operate, needs to be considered during system specification (Wieringa, Dubois, and Huyts 1997). The history of software engineering is littered with the remains of systems that did not meet their users' needs and did not help the enterprises, groups, or individuals for whom they were intended. Furthermore, the majority of software errors can be traced back to incorrectly specified requirements; however, the longer an error remains undetected, the higher will be the cost of fixing it (Boehm 1984). This is especially true when development is not traceable and its rationale has not been recorded. The question is how requirements engineering can help produce high-quality telecommunications software.

The vast complexity of telecommunications systems is characterized by the variety of technologies they employ and the many diverse participants in their development. A tremendous legacy has accumulated over time. This presents a very challenging task for service developers, in that it is simply impossible for a service designer to have an adequate understanding of all the factors to be considered during service development. What means can be found to relieve the designer of the need to master such a huge amount of detail? Although AI seems to be promising, little research has yet been undertaken into the use of AI for requirements engineering in telecommunications (Ryan 1994).

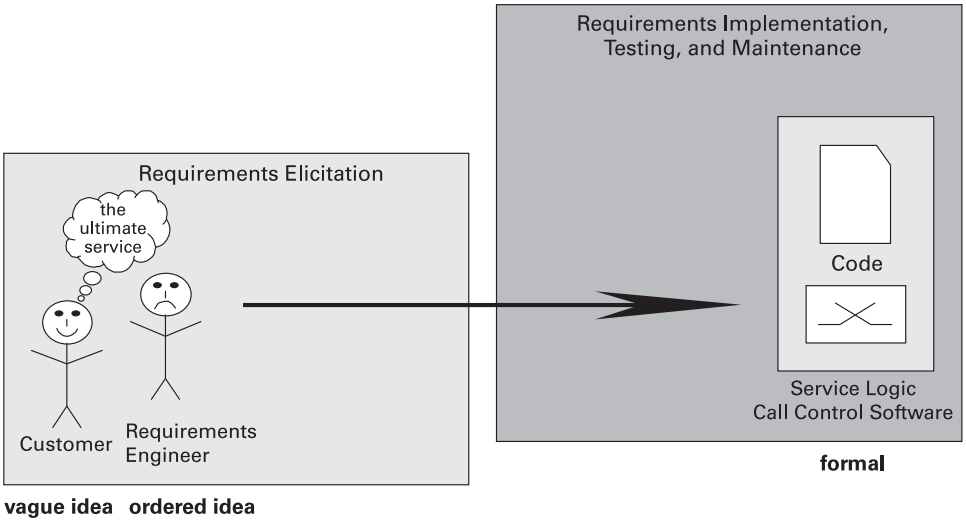


Figure 4.1
Traditional software development in telecommunications

Unfortunately, traditional ways of developing telecommunications software (see figure 4.1) have resulted in many challenges. Key among these challenges are long development time and feature interaction. Yet another challenge for developers is the great conceptual gap between an initial service idea and the software that eventually implements the service. Unfortunately, many telecommunications companies do not have good software processes in place to bridge this gap.

The research outlined in this section investigates the use of advanced software engineering, requirements engineering, and AI technologies for telecommunications service design. This research has led to the Requirements Assistant for Telecommunication Services (RATS), a service development methodology[1] with a supporting tool (the RATS tool). The methodology is based on a three-dimensional framework originally inspired by the NATURE project (Pohl 1994). However, the framework has been specialized for the process of development of telecommunications services, with each dimension addressing essential aspects of service development. Progress within this framework is achieved by following the RATS methodology guidelines, which on the one hand clearly instruct the service designer, but on the other hand leave the designer considerable freedom, as is necessary during development. The aim is a complete, refined, hierarchical, and formal service specification in the Specification and Description Language (SDL) as recommended by the International Telecommunication Union (ITU) (1999). The RATS tool has been implemented in the knowledge representation language Telos (Mylopoulos et al. 1990). This

language has been used for several purposes: modeling of the telecommunications domain and modeling of the RATS service development methodology, as well as the implementation of intelligent development support.

4.2 Usage of the Tool

The main users of the RATS tool are *requirements engineers* (REs) and *service designers*. These two groups of people use the tool during service development. The tool assists them during requirements acquisition, modeling, analysis, traceability, documentation, and specification. From the very beginning, all requirements are stored in the RATS tool. Although the current version of the tool handles only the early life cycle up to the formal specification of the telecommunications service (i.e., requirements and their management), the final aim is to provide assistance in all steps of the development process.

In addition to the requirements engineer and designer, *project managers* can also use the tool. The implementation of a requirements engineering life cycle model (i.e., the RATS methodology) in the tool allows the manager of a project to identify the current status of the project's development. As will be explained later, each stage of the development process has a unique label assigned to it; that is, the project manager can easily check in what stage of the RATS methodology the project development currently is.

The RATS tool contains too many technical details and telecommunications-specific terminology to be used by the *end user* of the telecommunication service. However, a browser interface could be developed that would allow browsing through the telecommunications domain models included in the tool. Since the tool contains information from different sources (more than eighty different telecoms' recommendations and standards, expert knowledge, and other telecommunications resources), it is an excellent aid for *new staff* to learn about the telecommunications domain. The current version of RATS contains information on various telecommunications networks (such as the Public Switched Telephone Network [PSTN] and Integrated Services Digital Network [ISDN]) and their associated services as well as customer premises equipment (CPE). Extension of the tool with more domain information on feature interaction, network interworking, switches, etc., is planned. This would enable new developers to learn about the telecommunications domain using conceptual models rather than "dry" standards.

4.3 Integration with Other Systems

The aim of the RATS tool is to provide a smooth transition from initial service idea to formal service specification in the ITU-recommended SDL. In order to gain max-

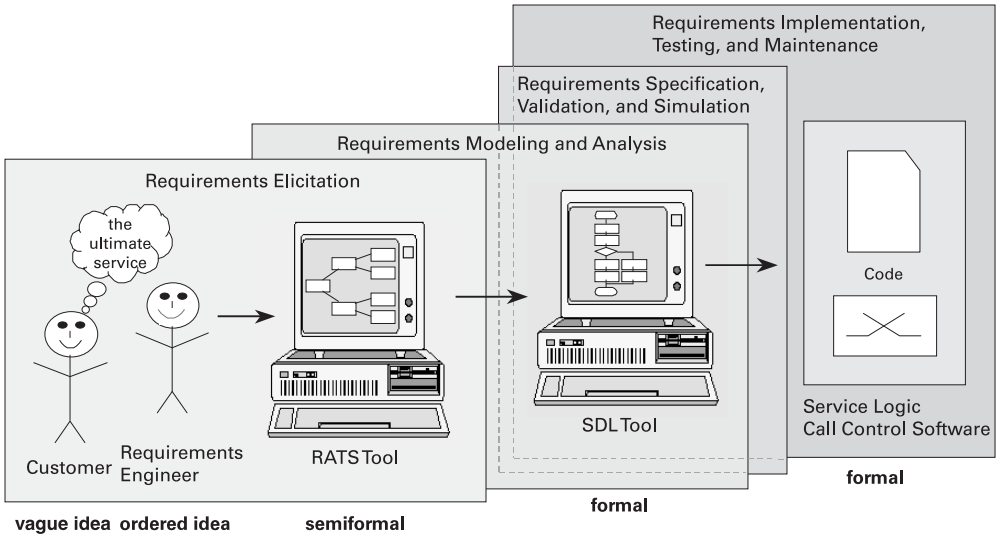


Figure 4.2
Service development using RATS

imum benefit from the methodology, the RATS tool should be used in combination with an SDL tool. Figure 4.2 shows a possible example development methodology that makes use of both the RATS tool and an SDL tool (such as SDT [SDL Design Tool] from Telelogic or Cinderella SDL from Cinderella).

When used in combination with an SDL tool, the RATS tool is used for the early phases of requirements engineering, during which high-level goals are collected and refined into lower-level requirements. The outcome of the refinement of the nonfunctional-requirements (NFRs) process is either implementation constraints or functional requirements. The functional requirements are then specified semiformally using a three-stage use case design process that leads to a use case notation aligned with SDL. The current version of RATS does not allow the automated export of this functional-requirements specification into SDL tools: The developer still needs to translate the final RATS specification into SDL. But it is envisioned that at least a first-cut translation can be automated.

Taking the semiformal specification produced by the RATS tool as input, the SDL tool is then used for architectural design and to produce a formal functional requirements specification. SDL is a very powerful language for the specification of reactive systems that can be defined with an extended state machine. Many SDL tools have powerful modeling, verification, validation, and animation tools that help improve the correctness and completeness of the functional specification. At this point in the system development, the influence of nonfunctional requirements on the

design has to be considered, and special care has to be taken that these requirements are satisficed.[2]

4.4 Main Functions of the System

The RATS tool is an expert system that is based on various conceptual models whose combination and interaction provide the tool user with active support during telecommunications service design. One of the aims of the RATS project has been to show that active development support is possible even in the intrinsically difficult part of the early system development life cycle. Currently, there is a lot of support for coding and testing and growing support for functional and architectural design, but only very limited support for requirements engineering. Commercial support for requirements engineering is at the moment limited to two categories of tools: modeling tools (such as TAU from Telelogic) and requirements management tools (such as DOORS from Telelogic and RequisitePro from Rational). There are currently no commercial tools that provide active support for the requirements engineering process. It is obvious that active support is much more difficult to provide in the early life cycle phases than in later phases. The objective of the RATS project was to demonstrate the concept that such support is possible. The RATS tool has established that active support is possible if a well-defined methodology is used (in this case, the RATS methodology).

However, there is a trade-off between the amount of support that can be provided and the genericness of the tool. A generic tool is not able to provide very specific guidance in the requirements engineering process. All currently available commercial requirements engineering tools (i.e., the previously mentioned modeling and requirements management tools) are in this category. However, the more domain specific a tool is, the more support it can provide. If a tool contains very detailed domain information expressed in conceptual models, specific guidance can be provided. But the disadvantage is that if domain information changes frequently, as is the case in a high-tech environment, maintenance of these models requires major effort.

The RATS tool has been implemented in such a way that guidance in the requirements engineering process is provided in several different ways in order to provide comprehensive assistance to the developer. Figure 4.3 shows the different categories of guidance available to the RATS tool user.

Two *kinds of guidance* are distinguished in the RATS tool:

- *passive guidance*, which points out mistakes in the specification
- *active guidance*, which tells the service designer what to do

Active guidance is again subdivided into the two levels for which guidance is provided:

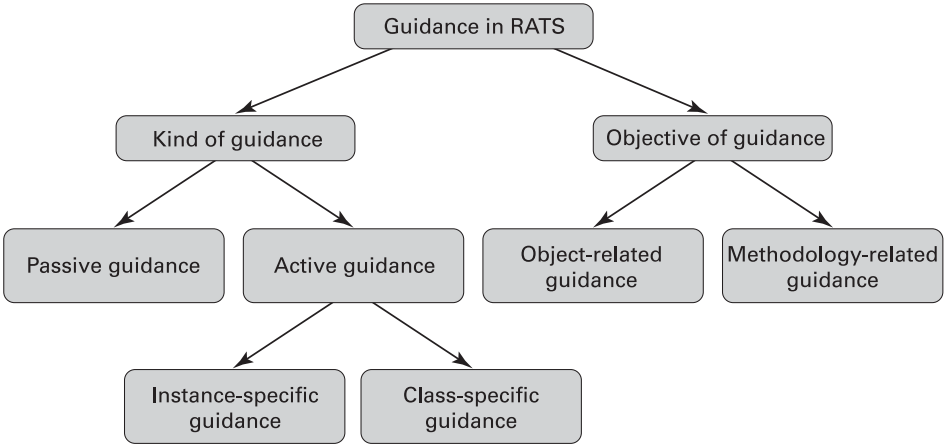


Figure 4.3
Categories of guidance in RATS

- *class-specific guidance*, which provides general help for the class level of the models
- *instance-specific guidance*, which provides guidance for individual instances of the classes

As can be seen in figure 4.3, the guidance offered by the RATS tool can be classified according to the *objective of guidance* as well as according to the kind of guidance. The low-level concepts of RATS (e.g., requirements, goals, documents) and the RATS development methodology can be viewed separately. As the abstraction levels of the basic concepts and of the methodology are very different, the following distinction is made according to the objectives of guidance:

- *methodology-related guidance*, which provides help with the RATS methodology
- *object-related guidance*, which provides help with the basic objects and concepts used in the RATS methodology

The implementation of these different kinds of guidance using conceptual models will be explained later in the chapter.

4.5 Architecture of the Tool

Before explaining some of the conceptual models used to construct the RATS tool in detail, I present the overall architecture of the RATS tool. The architecture is basically client-server, and the server has been implemented in ConceptBase. Since ConceptBase supports client-server communication via the Internet, the RATS client

can run on a different workstation or even a different site than the RATS server. This enables distributed requirements engineering. The ConceptBase tool is limited, however, in that a client may communicate with only one server at a time, but many clients may be connected to the same server. The present version of ConceptBase does not yet support concurrency control beyond the serialization of messages (Jarke, Jeusfeld, and Staudt 1999a).

4.5.1 The RATS Client

The RATS client is still under development. It is being implemented in Java; that is, it uses the Java interface provided by the ConceptBase team (Jarke, Jeusfeld, and Staudt 1999b). Although the ConceptBase workbench could be used as the interface to the RATS server, it would be very impractical, since even a very simple operation requires many interactions with the server. Additionally, not every user of the RATS tool can be expected to know the Telos language. For this reason, the main task of the RATS client is to improve the usability of the tool.

The RATS client consists of three parts (see figure 4.4):

- the graphical user interface
- the client logic
- the frame generator

4.5.1.1 The Graphical User Interface The user of the RATS tool (usually a requirements engineer) interacts with the RATS tool via the graphical user interface (GUI). The main tasks of the GUI are

- to display information to the user;
- to accept user input.

There are several considerations involved in the display of information:

- Information contained in the knowledge base of the RATS server needs to be displayed to the user in an easily understandable way. This information could include requirements and their attributes; the relationships between requirements; the contents of requirements documents, as well as their attributes; the contents of libraries contained in the domain models; and the progress achieved in the development process.
- The display needs to have means for filtering requirements and for displaying a specification at different levels of abstraction.
- Information that has been derived from the contents of the knowledge base, like the history of a requirement or the impact of a change in requirements, needs to be shown to the user.

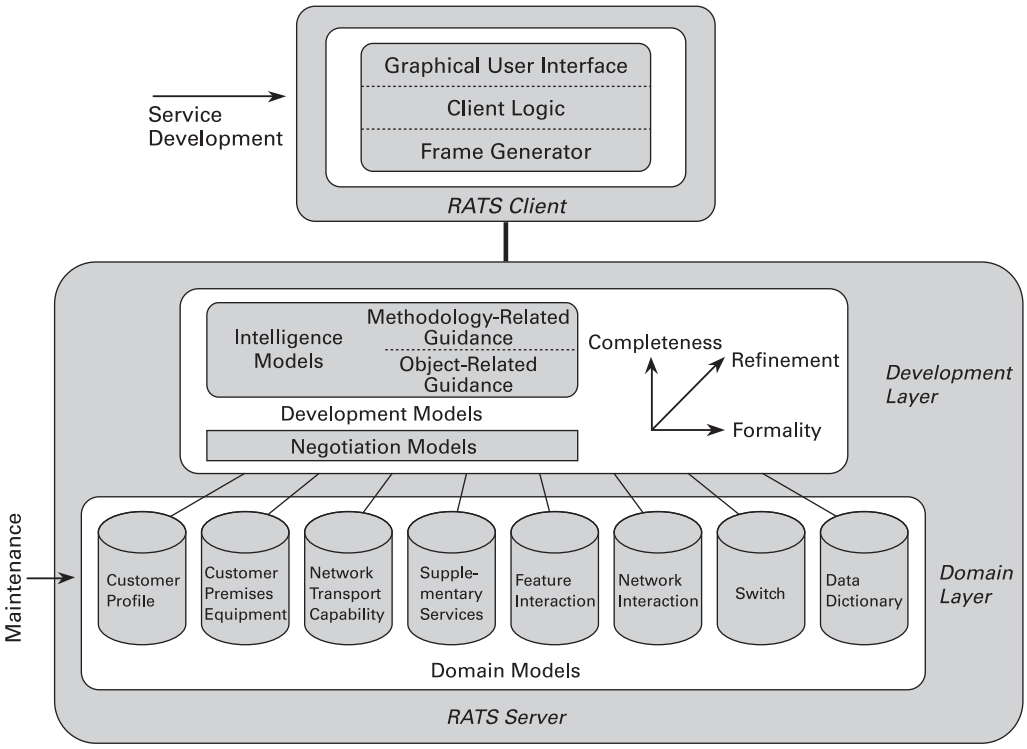


Figure 4.4
The RATS tool architecture

• The guidance given by the RATS tool needs to be displayed. This can be guidance related to a low-level object (like a specific requirement) or to the overall RATS methodology.

• Responses of the tool about the outcome of the operations that have been performed must be displayed to the user.

Forms are used for most user input. These forms are mainly determined by the templates of the underlying conceptual development models. The user has to complete and edit the forms before the desired operation on a requirement can be performed and the changes are made in ConceptBase.

4.5.1.2 The Client Logic The client logic (CL) is a transaction generator that mediates between the GUI and the frame generator. The GUI passes the input of the RATS user to the CL, which then decides how to deal with the user request. Having decided on the appropriate strategy for processing the user request, the CL calls methods of the GUI and the frame generator to achieve the requested operation.

To achieve a single user operation (e.g., create a new requirement) takes several transactions within the RATS tool. It is the task of the client logic to generate and coordinate the appropriate sequence of actions (like `ask`[3] the ConceptBase server, or `insert` (i.e., `tell`) and `delete` (i.e., `untell`) objects contained in the ConceptBase server) that achieve the desired user operation.

4.5.1.3 The Frame Generator The RATS server has been implemented in the Telos language, which means that all interactions with the server need to be performed using this language. Consequently, any user information and all transactions created by the client logic need to be translated into the frame notation of Telos; this is essentially the task of the frame generator (FG).

The frame generator has the additional task of performing the reverse operation, that is, stripping off a Telos frame and extracting the information to be displayed by the GUI. The frame generator therefore acts as a two-way translator whose actions are triggered by commands received from the client logic.

4.5.2 The RATS Server

The RATS server is implemented in ConceptBase and contains all the conceptual models expressed in Telos. The server is subdivided into two layers:

- the development layer
- the domain layer

4.5.2.1 The Development Layer The development layer of the RATS server contains the key components of the server. This layer is, divided yet again into three modules, each containing several conceptual models:

- the intelligence module, containing the intelligence models
- the development module, containing the development models
- the negotiation module, containing the negotiation models

Despite their different natures and tasks, all three modules are implemented in the Telos language and are closely linked with one another. Together, they contain the implementation of the overall RATS development methodology. The modules provide advice to the service designer during the development of a new telecommunications service and help prevent erroneous specifications. The Telos language is well suited to modeling the concepts involved in the RATS development methodology. Consistency can be ensured through Telos constraints, and Telos rules provide active guidance for the service designer. Later in the chapter, the implementation of these models is explained in more detail.

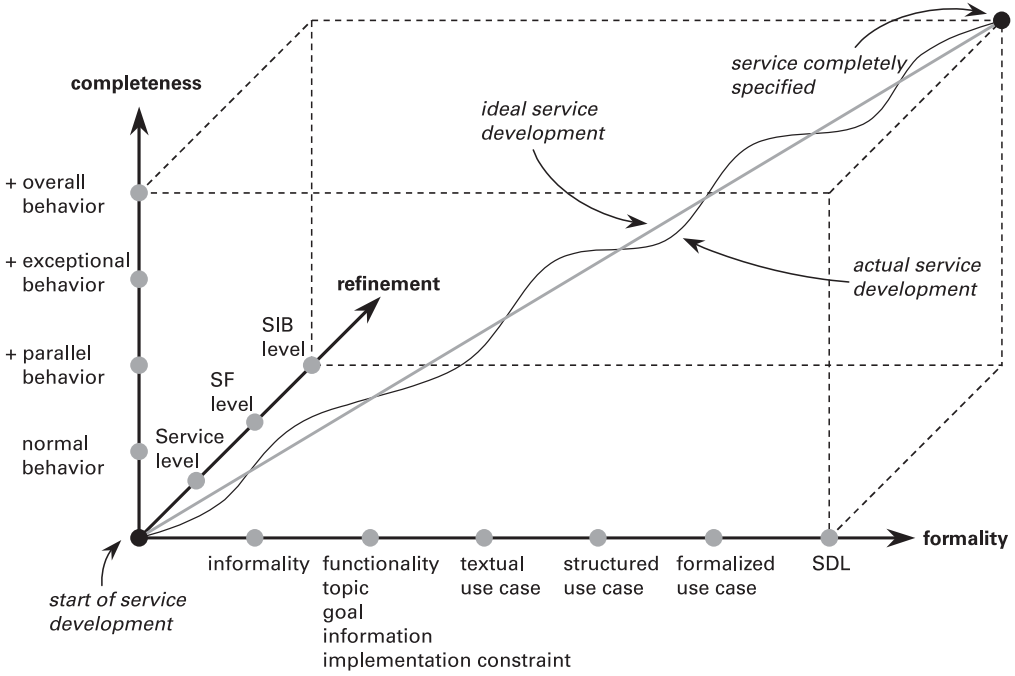


Figure 4.5
The three-dimensional framework of the service development process

The key component in the development layer is the methodology guidelines, which are derived from the underlying three-dimensional framework for requirements engineering for telecommunications services (see figure 4.5).

4.5.2.1.1 The Three-Dimensional Requirements Engineering Framework of RATS The RATS requirements engineering framework takes a "divide and conquer" approach. Complex processes are made more accessible by splitting them into smaller subprocesses that are, ideally, independent of one another. The framework has three dimensions:

- completeness
- refinement
- formality

The dimensions represent three major concerns during telecommunications service development, with each dimension having well-defined states and being independent of the other dimensions. In theory, this has the advantage of allowing one aspect of

telecommunications service development to be focused on, without having to consider the other dimensions at the same time. However, practice shows that one step in the development of a service usually involves progress in more than one dimension. For instance, a specification usually becomes more complete when requirements are refined, and formalizing a specification can help in detecting missing requirements, thus contributing to completeness.

Specifying states for each dimension enables the definition of milestones and metrics for the service development process. It also provides possibilities for assigning responsibilities to service developers, as well as for checking on the development process and its timeliness.

4.5.2.1.1.1 The Completeness Dimension Achieving a complete requirements specification is challenging. Considering that about a quarter of all changes to requirements are the result of requirements overlooked in the original specification process, it appears that we still are not able to do a good job of completely identifying system requirements. A commonly used approach to requirements gathering is to focus on different aspects of system usage at different times. The telecommunications service specification template described in ITU-T I.210 (ITU-T 1993b) recommends that the normal behavior of a service be focused on, then the exceptional and alternative behavior. Other software engineering methods take a similar approach.

The completeness dimension of the RATS framework also uses a similar approach. As can be seen in figure 4.5, the normal behavior, that is, the most common usage of the service, is specified first. Focusing first on the standard usage of a system helps developers to get the key features right. After the specification of normal behavior, parallel behavior is added. Exceptional behavior is added at the end and deals with possible errors. These steps use a top-down approach to development. The final step of the completeness dimension uses a bottom-up approach. The individual pieces of the specification are grouped together in order to define the overall behavior of the system. This grouping is performed in a hierarchical manner at different levels of abstraction.

4.5.2.1.1.2 The Refinement Dimension The refinement dimension of the RATS framework is geared toward the Intelligent Network (IN) architecture (Thörner 1994). However, any levels of abstraction can be chosen that are appropriate for a domain. The IN architecture uses three different planes: The *service level* is the highest level and contains stand-alone commercial offerings to which a telecommunications customer can subscribe. The *service feature* (SF) level contains service components that are reused during the construction of a service. Examples are call forwarding and abbreviated dialing. The *service-independent building blocks* (SIBs) are low-level components that are used to compose the functionality of a service feature. Authenticate and charge are examples of SIBs.

4.5.2.1.1.3 The Formality Dimension The formality dimension has the highest number of states (see figure 4.5) among the dimensions of the RATS framework. The reason for the high number of states is that the transition from complete informality to a formal specification, which is the job of the formality dimension, has always been a major challenge for developers. The formality dimension starts off with an informal statement from the customer. These informal initial requirements then have to be grouped into various subclasses: functionality, topic, goal, information, and implementation constraints. The transitions among these subclasses that makes sense during refinement are limited. For instance, it makes sense to refine a goal into functional behavior. However, it would not be appropriate to refine functional behavior into information. The motivation behind these constraints is the aim to generate more concrete requirements at a lower level of abstraction out of high-level, abstract, nonfunctional requirements. Inappropriate transitions are prevented by constraints that are defined in the conceptual Telos models. The functional requirements of the system are then specified using a three-stage use case design process. During the first stage, the functional behavior is specified by textual use cases, which are written in natural language in a relatively free style. Structured use cases, developed in the second stage, are an enhanced version of the textual use cases and contain additional information, such as pre-, flow, and postconditions. The most formal version of use cases are the formalized use cases employed in the third and final stage, which are geared toward the notation of SDL, the language recommended by the ITU for the specification of telecommunications systems.

4.5.2.1.2 The Methodology Guidelines The development layer of the RATS server also contains conceptual models derived from the RATS framework for requirements engineering (see figure 4.5). The key component of the development layer, the methodology guidelines, were developed based on the following assumptions:

- The initial starting point of service development is an incomplete, informal, high-level description of the desired service. The output aimed for is a complete, consistent, fully formalized and refined specification of the service.
- The ideal service development takes place along a straight line from the initial point in the development space to the desired end point. This has not been explicitly proved; however, real-life experience underlines the feasibility of this assumption.

Development along this "ideal" line will never be achieved in practice. As shown in figure 4.5, actual service development looks much more like a wavy curve oscillating around the ideal line. The task of the methodology guidelines is to ensure that the developer does not diverge too far from the ideal line but instead stays as close as possible to it. Table 4.1 describes, in a concise way, the actions to be performed by the service designer, step by step, during service development.

Table 4.1
RATS methodology guidelines for service development

Action		By	Reason	Document
For service level				
Outline service		Customer	Customer-centered requirements engineering	ICD
Brainstorming		Customer, REs	Encourage innovation, increase completeness	BL
Evaluate brainstorming list		Customer, REs	Increase agreement	BL
Categorize requirements		REs	Increase formality	BL
Define service		Customer, REs	Increase completeness	SDT
For service level and SF level				
Group requirements into self-contained functional blocks		REs	Preparation to find reusable functional blocks	SDT
Add parallel behavior	Action: Map functionality blocks onto existing ones from the library by: REs Reason: Reuse of existing functionality	Customer, REs	Increase completeness	SDT
Define behavior with textual use cases		Customer, REs	Increase formality	SDT
Refine and decompose nonfunctional requirements and satisfy them with textual use cases or implementation constraints		Customer, REs	Address nonfunctional requirements	SDT
Organize textual use cases into overall use cases of textual use cases		REs	Better overview, bottom-up approach, increase completeness	SDT
Define states in the behavior		REs	Improve structure, makes translation into structured use cases easier	SDT
Translate textual use cases into structured use cases		REs	Increase formality	SDT
Organize structured use cases into overall use cases of structured use cases		REs	Better overview, bottom-up approach, increase completeness	SDT
Add exceptional behavior		Customer, REs	Increase completeness	SDT
For SF level				
Translate structured use cases into formalized use cases		REs	Increase formality	SDT
Translate formalized use cases into SDL		REs	Increase formality	SDT/SDL Tool
SDL design		Service designers, customer	Formal analysis, verification, simulation, validation, testing, code generation	SDT/SDL Tool

Note: ICD: initial customer description; BL: brainstorming list; SDT: service definition template. Note that the "map functionality blocks" action goes on in parallel to the other actions.

Initially, the customer states the idea for the proposed service using a textual description. All development must be validated against this initial statement, the *initial customer description* (ICD). For instance, the customer might briefly state that it wants users to be able to make a phone call from a public phone box and have the call billed to their home telephones rather than use cash or a credit card. Without necessarily knowing the proper telecommunications terminology, the customer is asking for some kind of calling-card feature.

After writing the ICD, the customer conducts at least one brainstorming session, together with telecommunications specialists, in which topics and issues, such as authentication and authorization, are raised. A list of these topics and issues is compiled; this list is, initially, merely a collection of requirements, but later it will be evaluated. A subsequent categorization of the collected and evaluated requirements is performed by the requirements engineer; as it does not add content to the specification, there is no need to involve the customer in this step. This categorization process is necessary to increase the formality of the requirements through constraints and to provide active guidance to the requirements engineer during the process of refining the requirements. After this preliminary work, a template is used to define the proposed service in a more comprehensive way. This service definition template addresses all the topics and issues that must be considered in designing the new service. Both parties, the customer as well as the requirements engineer, have to complete the document that is developed using the service definition template.

By this point, the most common expected interactions between the system and its users should be specified. In the case of the calling-card feature, the basic service behavior is expected to be as follows: The user wants to make a phone call from a public telephone using a calling card. He or she first lifts the receiver and enters the access phone number on the keypad, then authenticates and authorizes himself or herself. Then he or she can dial the destination number, and a connection is established.

Once the normal behavior of the system has been specified, the requirements need to be grouped into functional blocks that are relatively independent of one another. Each functional block is equivalent to a feature or subfeature of the service being designed. This grouping into functional blocks is a first step toward the reuse of functionality. From this point on, the requirements engineer should constantly be evaluating whether some of these functional blocks can be mapped onto already existing service features or subfeatures.

With the help of the question "What else should the service be able to do?" alternative or parallel behaviors must be determined and specified. In the calling-card example, the user might want to phone another destination after the completion of the first call. Rather than having to hang up and reauthenticate and reauthorize, the user

should be able to press a key combination (e.g., ***) that allows him or her to place another call immediately upon finishing the previous one.

After the identification and specification of alternative behaviors, the use case design process starts. Up to this point, information about the functionality required in the system has mainly been collected without much of an effort to relate aspects of functionality to one another; relationships among these aspects are now identified as the system's functionality is specified with the help of textual use cases. This forces the customer and requirements engineer to go through all possible user-system interactions.

At this point, the nonfunctional requirements (NFRs) of the system need to be considered, as they have a major influence on the specification process by helping the customer and requirements engineer to decide among specification alternatives and to identify new system requirements. This is a long process that ends only when all the NFRs have been satisficed by either implementation constraints or functional behavior.

It is then time to look away from the details and see how far the development of the system has progressed, by organizing the textual use cases into a hierarchy of overall use cases. This not only shows any gaps that may exist in the specification but also gives a better overview of the service to be designed.

By this time a large number of specification blocks exist that need to be restructured in order to steer the specification in the direction of state transition notations. This not only is necessary for specifying the system in SDL but is also very helpful in relating the use cases to one another. To complete this restructuring, states have to be defined within the specified system behavior.

After the definition of states, the notation of structured use cases can be introduced; at this point, pre-, flow, and postconditions have to be considered. The whole collection of structured use cases then has to be organized into a hierarchical use case structure. This is a major milestone in the semiformal specification process, and at this point it has to be ensured that the specification is as complete as possible before any further development takes place.

One of the steps that still needs to be performed to increase completeness is to add exceptional behavior to the specification. Here, the behavior of the service during errors caused by the system or its users has to be outlined. The question "What can go wrong with this service?" will help reveal this behavior. In the case of the calling card, for example, the user might collapse in the phone booth after authentication and authorization. A time-out will then disconnect the user automatically after a specified period has elapsed.

It is questionable how much formality should be introduced into the specification and at what level of abstraction. Since formality is not much of an advantage for high layers of abstraction, RATS uses a higher degree of formality only for lower

levels of abstraction. Therefore, formalized use cases and SDL are employed only for service features and not for description of the top-level service.

Formalized use cases split the overall functional behavior of the system into minute chunks of functionality. Each use case contains a chain of *atomic actions* that describe very low-level behavior of three different kinds: input, output, and system operation. When the service has been completely specified using formalized use cases, the specification can be translated relatively easily into the graphical version of SDL with the help of a compiler.

At this point, the RATS methodology converges with existing SDL design methodologies.

4.5.2.2 The Domain Layer The domain layer of the RATS server consists of a comprehensive set of models describing the telecommunications domain as well as other areas relevant to telecommunications service development. The modules in this layer are created independently of one another and are like the development layer implemented in Telos.

The knowledge contained in the domain models is accessed by the negotiation models of the development layer during the service development process. The negotiation models help find reusable components (e.g., specifications of previously defined service features). Additionally, when used with a model browser, the domain models can provide general information about the telecommunications domain (e.g., information about network interoperation issues).

The current version of the RATS tool does not contain all the conceptual models shown in figure 4.4. However, the usefulness of the approach taken can already be seen from the currently implemented models, which contain comprehensive information from numerous sources. There is a great deal of scope for expanding this layer. The more comprehensive the domain models, the more active guidance can be offered to the service developer.

4.5.2.2.1 Kinds of Domain Models The information currently contained in the domain layer comes from various sources. The following list distinguishes three kinds of conceptual domain models according to their origin:

- *Standards* Domain knowledge is extracted from standards and expressed in conceptual models. The resulting standards-based models are very detailed and can be viewed as electronic versions of standards. When used with a model browser, they are suitable for reference and teaching purposes. However, such models usually contain more information than is necessary for service development and therefore slow down the performance of the RATS tool.
- *Expert knowledge* The knowledge of experts is captured and expressed in conceptual models. The resulting expert knowledge models are often very different from

those based on standards; however, they are one of the most appropriate models for the RATS tool, as they reflect real-world situations. Because they are created from the viewpoint of a domain expert, the structure of these models is usually well suited for requirements acquisition. Additionally, an expert describes only information that is relevant for service development, thus averting the inclusion of unnecessary information in the models.

- *"Quick models"* These models can be built from either standards or expert knowledge. They contain only the bare minimum of information that is absolutely essential to support the service design process and are restricted to the relevant domain objects, together with some characterizing keywords. This means that they can be created in a very short time, and their small size contributes to good tool performance.

With these three types, the RATS tool can accommodate very different kinds of domain models that might have been created by different people for different purposes. This allows flexibility and adaptation of the tool for different purposes.

4.6 Method Engineering

This section examines the conceptual models contained in the RATS server. These models are implemented in the Telos language and stored in ConceptBase. As has been mentioned previously, the RATS server contains the following layers and modules:

- the development layer, which consists of
- the intelligence module, containing the intelligence models
- the development module, containing the development models
- the negotiation module, containing the negotiation models
- the domain layer

4.6.1 The Intelligence Models

The intelligence models are those conceptual models of the RATS tool that contain the most advanced approaches to intelligence. The main task of the intelligence models is to assist the service designer by providing comprehensive guidance during service specification. This section describes how passive and active guidance have been implemented in the RATS tool. A combination of these two kinds of guidance leads to the overall intelligence of the RATS tool and helps to achieve the objectives of guidance: object-related and methodology-related guidance (see figure 4.3). Finally, the consistent use of libraries, as another task of the intelligence models, is outlined.

4.6.1.1 Passive Guidance

Passive guidance is basic, but essential, during service development. As the name suggests, passive guidance does not tell the service developer what has to be done but rather points out inconsistencies. Thus, passive guidance is a negative response of the tool in cases in which the user violates certain rules and constraints contained in the conceptual models.

Passive guidance can be achieved by using *rigid constraints*, which prevent the insertion of inconsistent objects into the knowledge base. Telos and its implementation in ConceptBase offer the following rigid constraints:

• *Object orientation* Conceptual, object-oriented modeling using formal languages, with their features of inheritance, classification, specialization, and instantiation, is a means of implementing some basic forms of intelligence. These features provide many constraints and restrictions on the definition of objects. For example, an individual instance can have only those attributes that it inherits from its class(es). Thus, any attempt to define an attribute that is not available on the class level will result in an error message from the tool, because the object-oriented axioms have been violated.

An example can be seen in figure 4.6. A formalized use case inherits all attribute categories from all superclasses; that is, an instance of `formalisedUseCase` also has a refinement status, requirements number, initiating actors, precondition, etc. Here is an example of an instance of `formalisedUseCase`:

```
Individual f967787432556 in formalisedUseCase with
  Summary,attribute
    summary : "Normal procedure of user access, identification and
    authentication procedure."
  PreCondition,attribute
    PreCon : "The system is idle."
  FlowCondition,attribute
    FlowCon : "User inputs the correct UPT service access code,
    UPT number and authentication code at the right time."
  PostCondition,attribute
    PostCon : "The user access, identification and authentication
    was successful. The system is ready for the identification of
    the procedure."
  attribute,containsAtomicActions
    aa1 : f937546783720
  attribute,involvedActors
    user : UPTUser
```

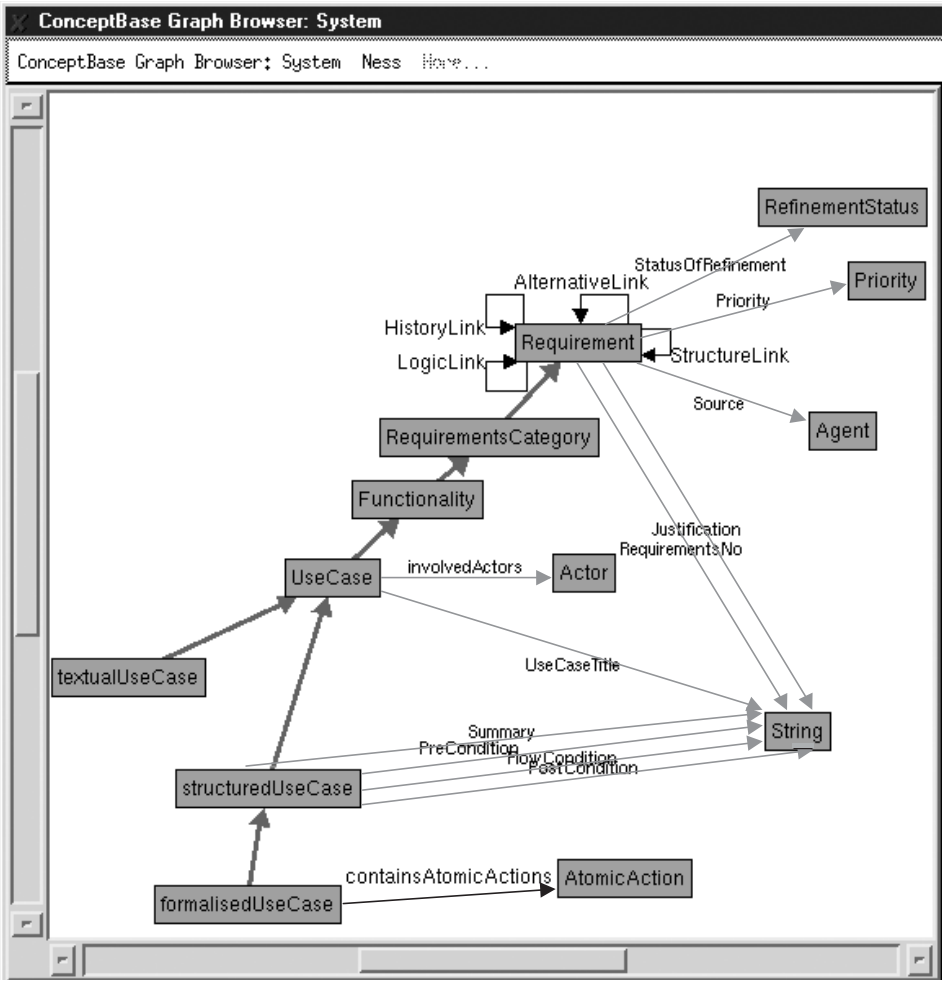


Figure 4.6
Passive guidance through object orientation. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

```
  StructureLink,attribute
    sl1 : f907546788765
end
```

However, this frame could not be inserted into the knowledge base if it were a textual use case, since instances of `textualUseCase` do not inherit the attributes of the classes `structuredUseCase` (e.g., attribute `PreCondition`) and `formalisedUseCase` (attribute `containsAtomicActions`). Thus object orientation helps in the correct handling of the different use case categories.

• *Telos axioms* Telos and its implementation in the ConceptBase tool contain several axioms, such as the naming axiom and the multiple generalization/instantiation axiom (see Jarke, Jeusfeld, and Staudt 1999a and Mylopoulos et al. 1990), that are automatically enforced as soon as an object is inserted into the knowledge base. The construction of Telos frames has to follow the syntactical rules of the language. This applies to the definition of objects as well as attributes. Any Telos frame that does not comply with these rules will be rejected.

• *Permanent user-defined constraints and rules* The various RATS models contain a large number of permanent, user-defined constraints and rules that ensure model consistency. The present version has over 70 user-defined constraints and 160 user-defined rules. An attempt to insert an object into the knowledge base that would violate any of these constraints is rejected. These rules and constraints are present in all modules of the RATS tool that have been implemented in Telos, that is, in the modules in the development layer and the domain layer. Although the constraints and rules can contain very complex formulas, from an AI viewpoint the constraints and rules are relatively basic AI concepts and part of virtually any Telos implementation.

An example of such a constraint is the following:

```
Individual GoalConstraint in Class with
  attribute,constraint
    GoalRefinement_con : $ forall f1/Goal f2/Requirement (f1
    LogicLink f2) ==> not((f2 in Topic) or (f2 in Information))$
end
```

This constraint addresses the issue of refinement of NFRs. As was mentioned earlier in the chapter, the transition from one NFR subclass to another has been restricted. This constraint ensures that an instance of `Goal` can only logically be refined into (i.e., have a `LogicLink` to) another `Goal` or `Functionality` or an `ImplementationConstraint`. The constraint is expressed in negative terms (i.e., `...==> not(f2 in...)` instead of `...==> (f2 in ...)`); otherwise it would implicitly enforce completeness, which is not wanted. This constraint is only meant to prevent the insertion of NFR refinements that are clearly against the rules.

Another example is given here:

```
Individual RequirementsObjectConstraint in Class with
  attribute,constraint
    Agreement_con : $ forall f1/RequirementsObject f2/
    RequirementsObject!StatusOfAgreement f3/Agent From(f2,f1) and
    (f2 RejectedBy f3) ==> not(f1 StatusOfAgreement Agreed)$;
    AcceptReject_con : $ forall f1/RequirementsObject f2/
    RequirementsObject!StatusOfAgreement f3/Agent (From(f2,f1) and
    (f2 RejectedBy f3) ==> not(From(f2,f1) and (f2 AgreedBy f3)))$
end
```

The constraint `Agreement_con` ensures that each instance of `RequirementsObject` can get the status `Agreed` only when no agent rejects that instance of `RequirementsObject`. The constraint `AcceptReject_con` ensures that an instance of `RequirementsObject` cannot simultanously be agreed on and rejected by the same agent.

These rigid constraints are extremely helpful because they immediately challenge the tool user when inconsistencies arise. However, they are desirable only for serious violations of consistency, such as syntactical or methodological errors. There are some inconsistencies, especially ones concerning incompleteness, that may be temporarily acceptable or even desirable (Balzer, Goldman, and Wile 1978). In order to accommodate these, there is a need for additional constructs that allow the temporary insertion of an object that is inconsistent or causes inconsistencies but that also indicate to the user that the object needs further attention. In RATS, such constructs have been termed *soft constraints*, and there are currently two categories of them:

• *Temporary user-defined constraints and rules* An additional set of user-defined constraints and rules, similar to the permanent constraints mentioned previously, is available. However, these are not constantly present in the knowledge base but are triggered only at certain milestones at which consistency and completeness must be checked. For instance, milestones in the development life cycle can be defined and can have a set of *check constraints* associated with them. If a service designer believes that a certain milestone in the service development has been achieved, the appropriate set of check constraints is temporarily inserted into the knowledge base to verify its consistency. Since milestones usually imply a certain degree of completeness, the constraints need to be designed in such a way that incompleteness manifests itself as inconsistencies. This means that many check constraints inspect a specification for missing information. If inserted in a successive manner, check constraints even allow the user of the tool to determine the location of each inconsistency and incompleteness in the specification that the constraints identify.

Here is an example of such a temporary constraint:

```
Agreement_con: $forall f1/RequirementsObject f1 StatusOfAgreement
Agreed)$
```

This constraint can be inserted into the knowledge base only if all requirements objects have been agreed on. If there is even one instance of `RequirementsObject` that does not have the status `Agreed`, the constraint will cause an error message. This makes this constraint a good example of a so-called check constraint.

• *Meta-attributes and state classes* Telos models can have several layers of instantiation. The higher the layer, the more generic the defined concepts. Attributes defined in high layers of the models can be seen as generic meta-attributes. Some of these meta-attributes can also be defined by metarules or metaconstraints. A meta-attribute defined by a metarule can be used to automatically assign an object to a *state class* depending on the current state of a specific object (e.g., a requirement). Statistical calculations of the distribution of objects in various state classes can provide the system designer with a valuable insight into the progress of the specification process.

To illustrate the functionality of meta-attributes and state classes, an example is given: In the top layer of the intelligence models a state class called `Inconsistent-DueToIncompleteness` has been defined. Additionally, there is a meta-attribute called `necessaryInc`, defined by the metarule `necessaryInc_rule`:

```
necessaryInc_rule: $ forall x/VAR
      (exists p/myc!necessaryInc c,d,m/VAR In(x,c) and P(p,c,m,d)
      and not(exists y/VAR In(y,d) and A(x,m,y)))
  ==> In(x,InconsistentDueToIncompleteness)$
```

The metarule `necessaryInc_rule` reads as follows: For any instance `x` of `c`, if `x` has an attribute link `p` with label `m` between the classes `c` and `d`, and `p` instantiates `necessaryInc`, and, in addition, if there exists no instance `y` of `d` that is the destination of the attribute link `p`, then the instance `x` becomes an instance of `Inconsis-tentDueToIncompleteness`.

The metaattribute `necessaryInc` can be used in all classes of the development models. For example, in order to enforce the definition of parallel and exceptional behavior as mentioned in the RATS methodology guidelines, the class `Functional-ity` has the following Telos definition:

```
Individual Functionality isA RequirementsCategory with
  attribute,necessaryInc
    Choice: Functionality;
    Exception: Functionality
end
```

This definition, in connection with the state class `InconsistentDueToIncompleteness` and the metarule `necessaryInc_rule`, automatically makes each requirement that is an instance of the class `Functionality` an instance of the class `InconsistentDueToIncompleteness`, if it has no parallel and/or no exceptional behavior defined. With the help of Telos queries, either all the instances of the class `InconsistentDueToIncompleteness` (i.e., all requirements that still have an incomplete definition) can be found, or the RATS tool can be asked whether a specific requirement is still incomplete.

Finally, the GUI can use state class information when displaying instances (e.g., a requirement) on the screen, so that the user can better oversee the process. For instance, the icon used for an incomplete requirement could be displayed in red, indicating that the requirement needs further attention.

The ideas described in this section have dealt with passive guidance, which is very helpful during service development. However, its capabilities are limited. It can only point out errors by stating: "Don't do that!" What is more valuable is showing the user what needs to be done next: "Do this!" The next section deals with this topic.

4.6.1.2 Active Guidance A very important characteristic of an expert system is the ability to give advice actively to the user of the system tool. There are several means by which active guidance can be provided using the Telos language, and some of the possibilities are outlined here. The first two types of guidance are defined on the class level; they thus provide class-specific guidance for certain classes, but they do not provide guidance for instances of the classes.

- *Strings assigned to classes* Classes contain strings that point out the next steps that normally have to be performed with the instances of the various classes. This information is similar to that provided by a design manual, which says what steps usually need to be taken for a certain object. For instance, the class `InitialCustomerDescription` contains a string called `description1` describing what needs to be done with the instances of this class. Such strings are usually textual descriptions of actions to be performed.

```
Individual InitialCustomerDescription in Class isA
RequirementsDocument with
  attribute
    Brainstorming : BrainstormingList;
    CreatedBy : Agent;
    AffiliationOfCustomer : String;
    PlaceOfCreation : String
  attribute,description
```

```
    description1 : "This description contains a high level
    overview/summary of the product. This will be a textual
    description of the product and should be very brief (max. 1/2
    page). It is written solely by the customer and expresses his
    ideas about the product to be developed."
end
```

• *Guidance derived from models* Some rough guidance can be derived from the way in which the models have been designed. However, this approach to providing guidance demands models that are oriented toward the development process rather than the conceptual inheritance of characteristics. In some cases, these two orientations might be the same; however, this cannot generally be assumed. In those cases in which these two orientations differ, the models would become very complicated if they additionally included guidance. The RATS development models have been created to give priority to conceptual inheritance rather than the development process. This means that only a very small amount of guidance can be derived from the development models. One example in which this principle has been used is the class `RequirementsObject` contained in the development models. `RequirementsObject` has an attribute called `Action`, which points again to the class `RequirementsObject`. This means that any instance of `RequirementsObject` must be transformed by an `Action` into a new instance of `RequirementsObject`.

These two types of active guidance are related to the class level of the development models and thus provide class-specific guidance, which can be very helpful as general advice. However, class-specific guidance gives the same advice for each instance of a certain class and does not deal with the development needs particular to of an individual instance, such as a particular use case or a brainstorming list. It is most helpful if the expert system is able to point out very specifically to the requirements engineer what should be done with a particular instance in order to proceed with its development. Therefore, more advanced approaches to active guidance offer instance-specific guidance with the help of intelligence models:

• *Intelligence rules and intelligence objects* Intelligence rules assign intelligence objects to specific instances of any class, depending on the actual state of the particular instance. The intelligence objects contain the necessary information to further develop the instance concerned. Intelligence rules can be very complex; however, they allow sophisticated, instance-specific active guidance. Using a generic query, all the intelligence objects assigned to a particular instance can be retrieved.

The functionality provided by intelligence rules and intelligence objects is illustrated by two examples. The first example looks at object-related guidance and is shown in figure 4.7. Depending on the development state of the specification of the Universal Personal Telecommunication (UPT) service, the object-related intelligence

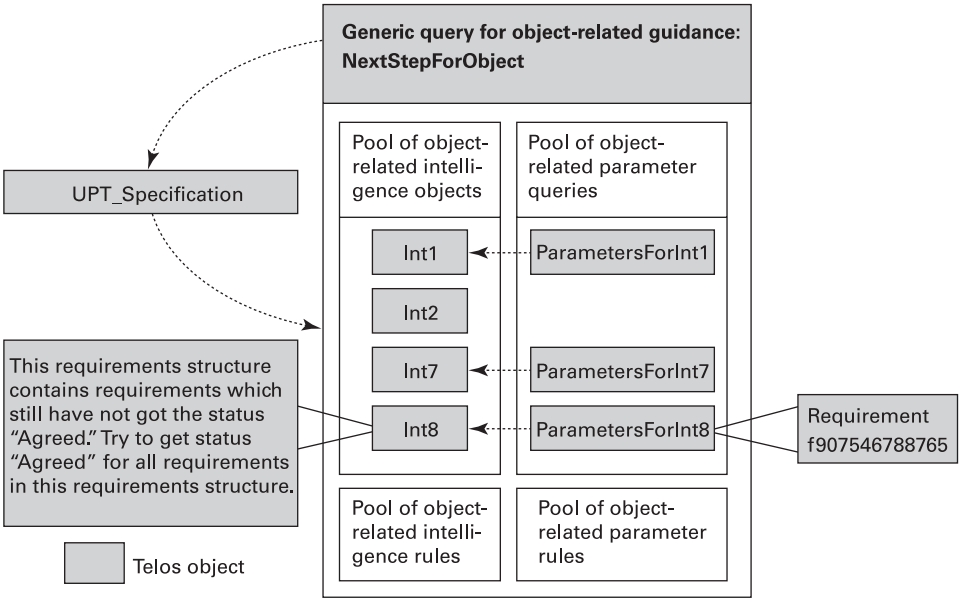


Figure 4.7
Example of the implementation of instance-specific, object-related active guidance

rules automatically create links from the instance `UPT_Specification` to the relevant object-related intelligence objects. These links are updated immediately by the object-related intelligence rules as soon as a change takes place in the RATS server. The intelligence rule that links `Int8` to `UPT_Specification` in this example is relatively simple:

```
Individual RequirementsStructureIntelligence in Class with
  attribute,rule
    ToDo1_rule : $forall f1/RequirementsStructure (exists e1/
    Requirement (f1 contains e1) and not(e1 StatusOfAgreement
    Agreed)) ==> (f1 IntelligenceObject Int8)$
  attribute,comment
    comment1 : "The 'ToDo1_rule' assigns the Intelligence 'Int8'
    to all requirements structures which contain requirements
    which are not all agreed on."
end
```

If a user of the RATS tool would like to know what needs to be done with regard to the current version of `UPT_Specification`, he or she asks the RATS tool for advice, using the generic query for object-related guidance `NextStepForObject` together with the parameter `UPT_Specification`:

```
Individual NextStepForObject in GenericQueryClass isA
ObjectRelatedIntelligence with
  attribute,parameter
    obj : ObjectWithIntelligence
  attribute,constraint
    con : $(~obj IntelligenceObject this)$
  attribute,comment
    comment1 : "This query finds the Instances of
    ObjectRelatedIntelligence which are relevant for the concerned
    ObjectWithIntelligence (obj)."
end
```

This query retrieves all those intelligence objects that are linked to the `UPT_Specification`. In this example, it is assumed that the UPT service definition is nearly complete; just one requirement has not yet been agreed on by all stakeholders. This causes the object-related intelligence rule to link the intelligence object `Int8`, defined as follows, to the `UPT_Specification`:

```
Individual Int8 in ObjectRelatedIntelligence with
  ToDo,attribute
    ToDo1 : "This requirements structure contains requirements
    which still have not got the status 'Agreed'. Try to get
    status 'Agreed' for all requirements contained in this
    requirements structure."
end
```

Using the generic query for object-related guidance, the service designer is asked to get the status `Agreed` for all requirements in the UPT service definition requirements document.

The second example explains methodology-related guidance, which helps with progress in the overall RATS methodology. The implementation is illustrated in figure 4.8 and is similar to that for the object-related guidance described previously. Whereas object-related guidance is concerned with low level issues, methodology-related guidance deals with high level methodological issues. It assists with overall service design, guiding the designer through the process of developing the whole service. Depending on the development state of the UPT service, methodology-related intelligence rules create links from the instance `UPT` to the relevant methodology-related intelligence objects. In the example of figure 4.8, it is assumed that the following rule assigns `Int1006` to the `UPT` object:

```
Individual FunctionalBlockIntelligence in Class with
  attribute,rule
```

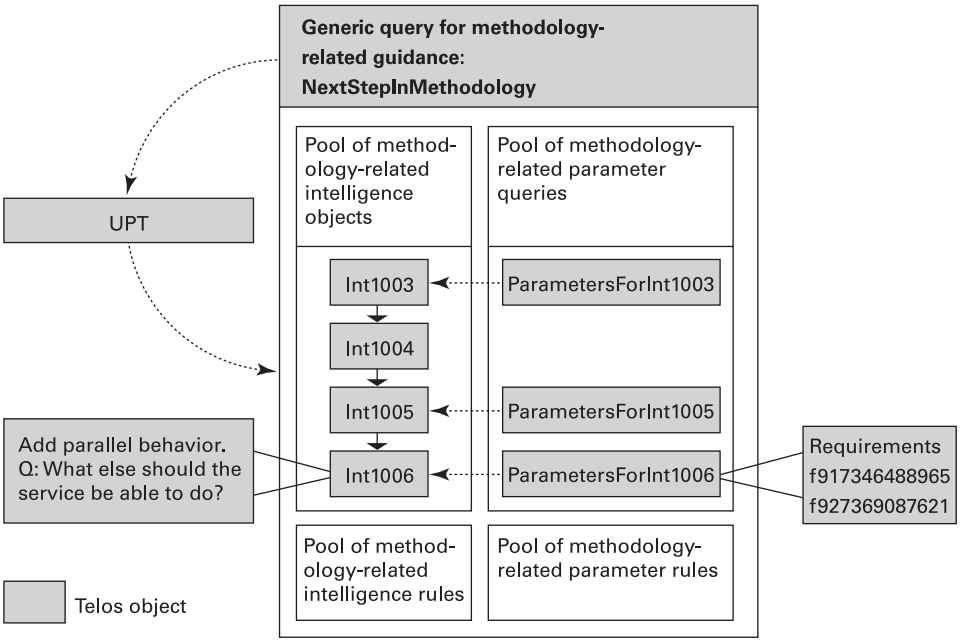


Figure 4.8
Example of the implementation of instance-specific, methodology-related active guidance

```
    AddAlternativeBehaviour_rule1 : $forall f1/FunctionalBlock
    (exists e1/Specification e2/Functionality (f1 definedIn e1)
    and (e1 contains e2) and not(e2 in RejectedRequirement) and
    not(exists e3/Functionality (e2 Choice e3))) ==> (f1
    IntelligenceObject Int1006)$
  attribute,comment
    comment2 : "The 'AddAlternativeBehaviour_rule1' assigns the
    Intelligence 'Int1006' to all Functional Blocks (i.e. Service,
    SF or SIB) which still contain functional requirements which
    have no alternative behaviour assigned to them."
end
```

`Int1006` is the intelligence object that asks the designer to specify alternative behavior:

```
Individual Int1006 in MethodologyRelatedIntelligence with
  ToDo,attribute
    ToDo1 : "The service designer has to add alternative
    behaviour. Q: What else should the service / service feature
    be able to do?"
```

```
  attribute,nextStep
    nextStep1 : Int1007
end
```

If the tool user wants the tool to tell him or her what has to be done next in order to progress in the RATS methodology, he or she can pose the following query:

```
Individual NextStepInMethodology in GenericQueryClass isA
MethodologyRelatedIntelligence with
  attribute,parameter
    obj : FunctionalBlock
  attribute,constraint
    con2 : $not(exists e1/NextStepsInMethodology[~obj/obj] (this
    doFirst e1))$
  attribute,comment
    comment2 : "This query finds exactly the next step which needs
    to be done according to the methodology for a certain instance
    of FunctionalBlock (i.e. for an Instance of Service). It does
    need the query NextStepsInMethodology and needs the rule
    MethodologyRelatedIntelligenceRule!doFirst_rule in order to
    work. This query is very slow."
end
```

This query finds exactly the next step that has to be performed. In order to find the *one* intelligence object that describes the next step, this query uses another generic query called `NextStepsInMethodology` that retrieves *all* intelligence objects that are linked to the functional block `UPT`. The query `NextStepsInMethodology` is defined as follows:

```
Individual NextStepsInMethodology in GenericQueryClass isA
MethodologyRelatedIntelligence with
  attribute,parameter
    obj : FunctionalBlock
  attribute,constraint
    con1 : $(~obj IntelligenceObject this)$
  attribute,comment
    comment1 : "This query does not necessarily make too much
    sense. It can come up with several instances of
    MethodologyRelatedIntelligence. It mainly serves as help for
    the query NextStepInMethodology."
end
```

One difference between intelligence objects that belong to the object-related guidance and those that belong to the methodology-related guidance is that the latter

ones are linked with one another (compare figures 4.7 and 4.8). The query `NextStepInMethodology` selects the very first intelligence object in the chain of intelligence objects that have been returned by the query `NextStepsInMethodology`.

- *Parameter rules* and *parameter queries* Intelligence objects can be associated with parameter queries. These queries contain complex rules that enable them to ask for specific parameters that help with further development of the service specification. Since these queries are linked to a particular intelligence object, which is linked in turn with a particular instance, the outcome of the query is specific to that particular instance and to the stage of development of that particular instance.

Going back to the example described in figure 4.7, the RATS tool will inform the service designer that there is the object-related parameter query `ParametersForInt8` connected to the intelligence object `Int8`. This query allows the user to ask the tool what parameters there are that are related to the UPT service definition. With the help of object-related parameter rules, the query `ParametersForInt8` finds those requirements contained in the UPT service definition that have not yet been agreed on by all stakeholders. In this case, there is only one requirement, the one with requirements ID `f907546788765`. Having received this information, the service designer can then proceed by again asking the tool what needs to be done to complete the requirement `f907546788765`. This new query and its associated parameter queries will then point out the names of all those stakeholders who have not yet agreed to this requirement and also why they have not agreed. This provides the service designer with the information necessary to proceed with the definition of the UPT service.

As can be seen in figure 4.8, there are similar parameter queries and rules for methodology-related guidance. These parameter queries will retrieve those objects that need further work. In the example shown in figure 4.8, two requirements are displayed that still need alternative behavior specified. Applying now the generic query for object-related guidance, the tool user can get additional instructions regarding the further development of these two requirements.

4.6.1.3 Intelligence Integration This section brings together the different aspects of intelligence. The previous example of the UPT service is used here as well to show how the different approaches to intelligence previously described work together in the RATS tool to achieve comprehensive overall intelligence. It should be noted that the various contributions to overall intelligence are distributed across all the layers of the RATS tool.

Assume again that the UPT service is to be defined in the requirements document called `UPT_Specification`. First of all, object orientation and Telos axioms ensure that the definition of `UPT_Specification` is an instance of the class `Specification` (which is a subclass of `RequirementsDocument`). This allows the `UPT_`

`Specification` to inherit all the attributes defined in the class `Specification`. Some of these attributes represent predefined headings that show the tool user the issues to be addressed during service definition.

It is further assumed that the development of the UPT service has already reached the point at which all functional behavior has been expressed in textual use cases. Meta-attributes and state classes will assign all requirements and documents that have not yet been completely developed to the class `InconsistentDueToIncompleteness`, which is a subclass of `InconsistentObject`. The GUI clearly indicates such objects by highlighting them. In order to show the service designer what to do next, *methodology-related intelligence rules* will find the current state of the development cycle and, with the help of *intelligence objects*, point out the next steps to be completed in order to progress in the specification of the UPT service. In this example, the tool will tell the designer to refine and decompose the nonfunctional requirements and to satisfy them with textual use cases and/or implementation constraints. *Methodology-related parameter rules* and *parameter queries* will list all those NFRs that require further decomposition at this point in the development process. Assuming that a certain goal needs to be refined further, the designer can now ask the RATS tool what needs to be done to achieve the goal. *Object-related intelligence rules* and *intelligence objects* will tell the designer that the goal needs to be refined into further goals, functionality, or implementation constraints. If the designer ignores this recommendation and tries to refine the goal into a topic, *permanent user-defined constraints* will reject this attempt.

4.6.2 The Development Models

4.6.2.1 The Overall Methodology The development models of the RATS tool contain the implementation of the RATS methodology, the theoretical framework of which is shown in figure 4.5 and the guidelines for which are described in table 4.1.

It seems to be sensible to define *states* (or *milestones*) within the RATS development process, with one *action* to be performed as long as the development process is in a particular state. This means that each action (i.e., each row) in table 4.1 represents a state in the development life cycle. In order to describe the actions that the service designer has to perform within each state, a methodology-related intelligence object has been defined for each state. Thus, the RATS methodology guidelines are implemented by linking each action of table 4.1 to a state, which in turn is associated with a methodology-related intelligence object describing the action to be performed as long as the development process is in that state.

The current state of the development process is determined by methodology-related intelligence rules, which link the service to be developed with the appropriate methodology-related intelligence objects. Since several of the methodology-related

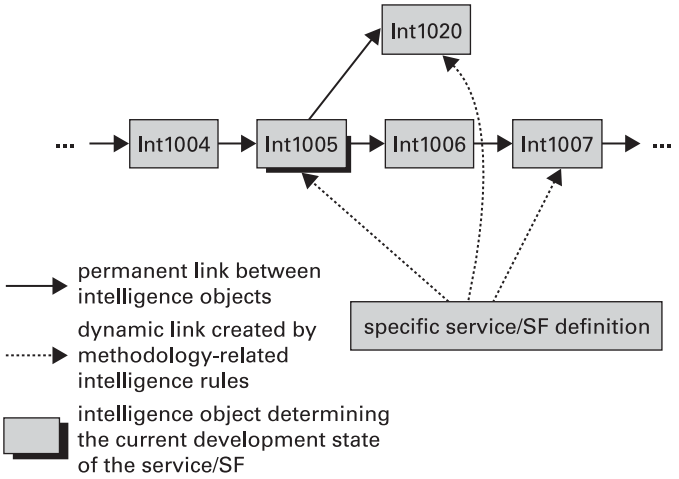


Figure 4.9
Determination of the current development state of a service or SF

intelligence rules might be true at a certain point in time, a service can have several intelligence objects associated with it (see figure 4.9). For instance, during the initial specification, in which only the basic service behavior is outlined, the rules for the two states "Add alternative behavior!" and "Add exceptional behavior!" are both true. The intelligence objects that are linked to a particular service are retrieved by the previously described generic query `NextStepsInMethodology`. In order to unambiguously assign a service to one single state, the states are sequentially linked with one another in the same order in which they occur during development (see figures 4.8 and 4.9). The earliest of all the states assigned to the service is then the current development state of the service. RATS determines the current development state by using the query `NextStepInMethodology`, which in turn calls the query `NextStepsInMethodology`. Actions that can be performed in parallel with other actions (see table 4.1) cause branching within the chain of intelligence objects (like `Int1020` in figure 4.9).

Transitions from one state to the next are achieved by performing appropriate actions (see figure 4.10). An important benefit of this approach is automatic *process iteration*. If, for instance, a service has already been specified using structured use cases (i.e., `Int1012` is now linked to the service, which means that service development is now in state 1012), and the customer adds a new requirement to the specification, rules will automatically initiate reengineering of the service from the appropriate point of the life cycle. Depending on the kind of change the customer is making, reengineering will start at a particular stage in the life cycle (see figure 4.11),

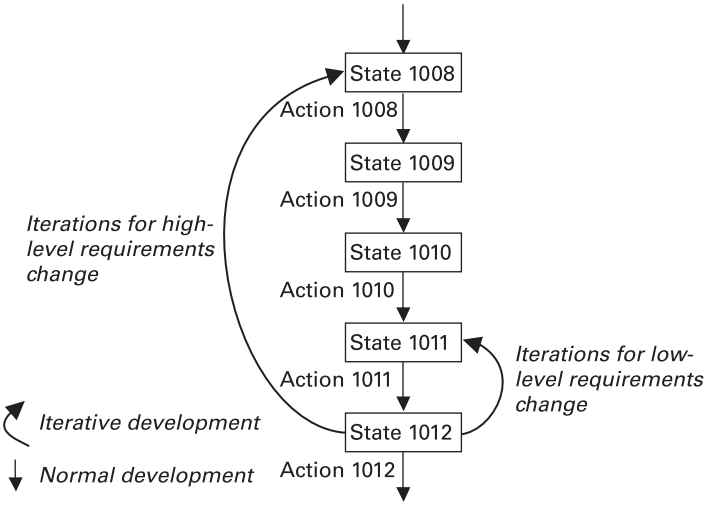


Figure 4.10
Development process iterations

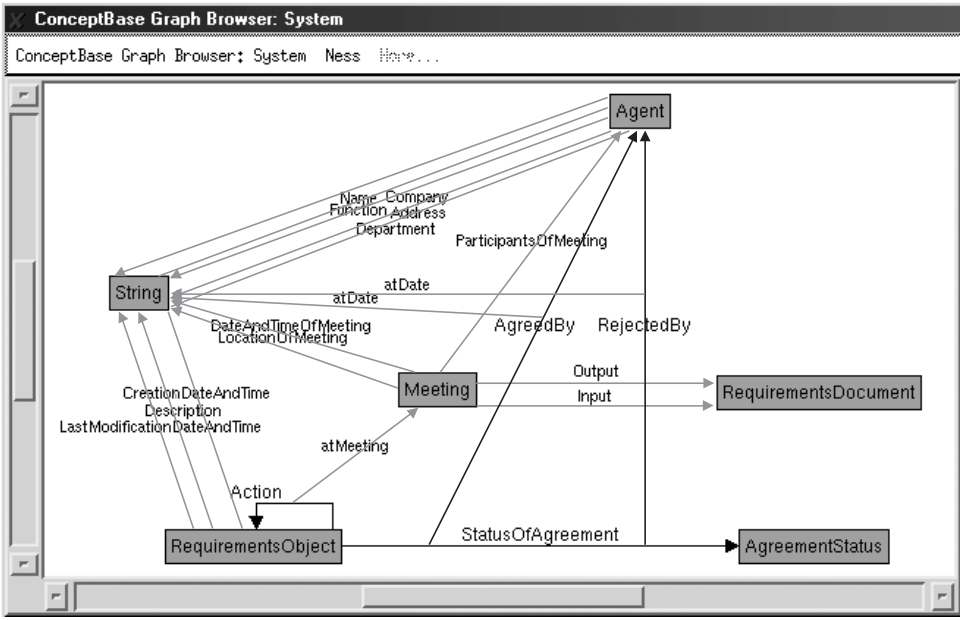


Figure 4.11
Excerpt from the metamodel of the development models. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

and that stage may be different for different types of changes. Adding a high-level nonfunctional requirement to an already existing requirements specification will cause a fallback to a very early state (in this case, state 1008), whereas adding minor low-level functionality (e.g., change of an announcement) will set the development process back by only a few states (in this example, to state 1011). This automatic process iteration to the nearest possible state in the life cycle is an excellent means of ensuring that reengineering effort is kept to a minimum when requirements change.

4.6.2.2 Implementation of the Requirements Engineering Concepts The development models contained in the development layer of the RATS tool (see figure 4.4) contain the implementation of the requirements engineering concepts used in the RATS methodology. This section focuses on the implementation of these concepts in object-oriented class hierarchies. The models comprise an abstract layer of classes and attributes that define generic characteristics that are used by many more-specific objects. One of the most generic concepts is modeled by the class `Requirements-Object` (see figure 4.11). A requirements object is such a general concept that any requirement (e.g., a use case, a goal, a service feature) and any requirements structure (e.g., a specification, an overall use case, a service) is an instance of the class `RequirementsObject` and thus inherits all its attributes.

Some of the attributes of the class `RequirementsObject` can be seen in figure 4.11. Any requirements object is annotated with the date and the time of its initial creation (attribute `CreationDateAndTime`) and its last modification (attribute `LastModificationDateAndTime`). Requirements objects can be linked with one another via the attribute `Action`. Links of this type are usually related to actions performed on a requirements object (e.g., decomposition of a requirement) and are created during meetings. Each meeting is annotated with its date, time, and location, together with the names of the people participating. A meeting additionally results in a change to the requirements documents; for instance, a brainstorming list is used as starting point for a meeting during which the service being developed is more precisely specified in a service definition document. The participants at the meeting (modeled by the class `Agent`), as well as their affiliations, are recorded.

The agreement status of a requirements object is very important, and therefore each requirements object needs to have exactly one agreement status assigned to it. This is achieved by combining two meta-attributes for the definition of the agreement status:

```
attribute,necessaryInc,singleCon
  StatusOfAgreement: AgreementStatus
```

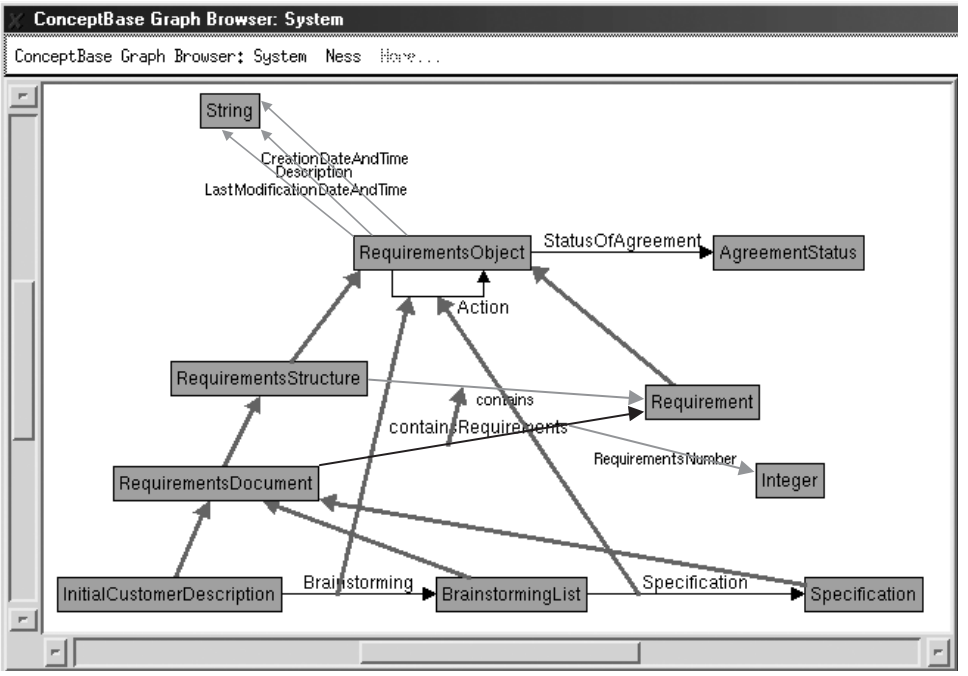


Figure 4.12
Excerpt from the `RequirementsDocument` subclass hierarchy. Attribute link ($\xrightarrow{\text{label}}$), subclass link (⟹), instance link (⟶), Telos object (▭)

- The `necessaryInc` meta-attribute marks each requirements object as incomplete if it has no attribute `StatusOfAgreement` defined. This meta-attribute belongs to the soft constraints of the RATS tool and allows temporary incompleteness.
- The `singleCon` meta-attribute ensures with a rigid constraint that this attribute is instantiated only once.

The attribute `RequirementsObject!StatusOfAgreement` itself has two attributes: `AgreedBy` and `RejectedBy` (see figure 4.11). These two attributes allow the user to list the opinions of the stakeholders regarding a particular requirements object. In order to see if a requirements object has been changed since a stakeholder (dis)agreed with it, the date of (dis)agreement is annotated as well (see figure 4.11). If the `LastModificationDateAndTime` is more recent than the date of (dis)agreement of stakeholders, then it might be worth pointing out to them the change in the requirements object and seek their approval of the change.

4.6.2.2.1 Requirements Documents Requirements documents are implemented in the RATS methodology as a special requirements structure, which is in turn a

Table 4.2
Assignment of `BrainstormingList` attributes to class attributes

`BrainstormingList` attribute	Class attribute
Name of service	First part of the name of the `BrainstormingList` object defined as instance of the class `BrainstormingList` or name of instance of class `Project` that has a `BrainstormingList` attribute to the `BrainstormingList` object
Date and time of creation	`RequirementsObject!CreationDateAndTime`
Date and time of last modification	`RequirementsObject!LastModificationDateAndTime`
Created in meeting	`RequirementsObject!Action!atMeeting`
Location of meeting	`Meeting!LocationOfMeeting`[a]
Date and times of meeting	`Meeting!DateOfMeeting`, `Meeting!StartTimeOfMeeting`, `Meeting!StopTimeOfMeeting`[a]
Participants of meeting	`Meeting!ParticipantsOfMeeting`[a]
Affiliation of participants	`Agent!Name`, `Agent!Function`, `Agent!Company`, `Agent!Department`, `Agent!Address`[b]
Status of agreement	`RequirementsObject!StatusOfAgreement`
Agreed by	`RequirementsObject!StatusOfAgreement!AgreedBy`
Rejected by	`RequirementsObject!StatusOfAgreement!RejectedBy`

a. The appropriate meeting is found via the attribute `RequirementsObject!Action!atMeeting`.
b. The appropriate participants are found via the attributes `RequirementsObject!Action!atMeeting` and `Meeting!ParticipantsOfMeeting`.

requirements object (see the subclass hierarchy presented in figure 4.12). This means that instances of the class `RequirementsDocument` inherit all the attributes, the constraints and rules defined for the classes `RequirementsStructure` and `RequirementsObject`.

The definition of the class `RequirementsStructure` again uses the `necessaryInc` meta-attribute as a means of allowing gradual development by tolerating temporary incompleteness during the definition of requirements structures (i.e., also of requirements documents):

```
Individual RequirementsStructure isA RequirementsObject with
  attribute,necessaryInc
    contains: Requirement
end
```

In order to illustrate in more detail how attributes are inherited within the object-oriented subclass hierarchy, the relatively simple example of the class `BrainstormingList` is given in table 4.2. This table shows the links between the `BrainstormingList` attributes as they are displayed to the tool user and the class attributes with which they are implemented.

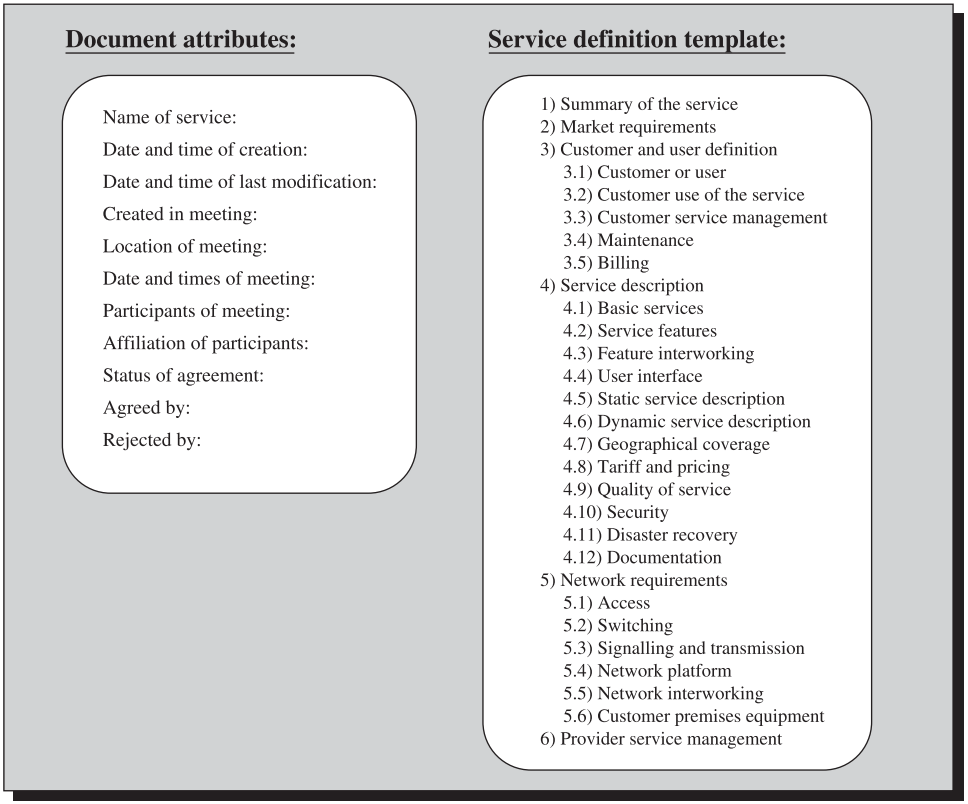


Figure 4.13
The service definition template

The attributes of the class `Template` determine the headings in each template. For example, the attributes of the class `ServiceDefinitionTemplate` reflect the main headings of the template used for telecommunications service development. Subheadings are implemented as attributes of the main headings. Before a new telecommunications service is specified, the template is copied; that is, the `ServiceDefinitionTemplate` is stored under a new name (e.g., `UPT_Specification` for the specification of the Universal Personal Telecommunication service) and made an instance of the class `Specification` instead of `Template`. Figure 4.13 shows the complete list of attributes, and figure 4.14 illustrates parts of the implementation. Those attributes that establish links between the requirements contained in the specification have their own attribute of the category `RequirementsDocument!containsRequirements!RequirementsNumber`. This attribute is used to construct the requirements number of each requirement in the requirements document.

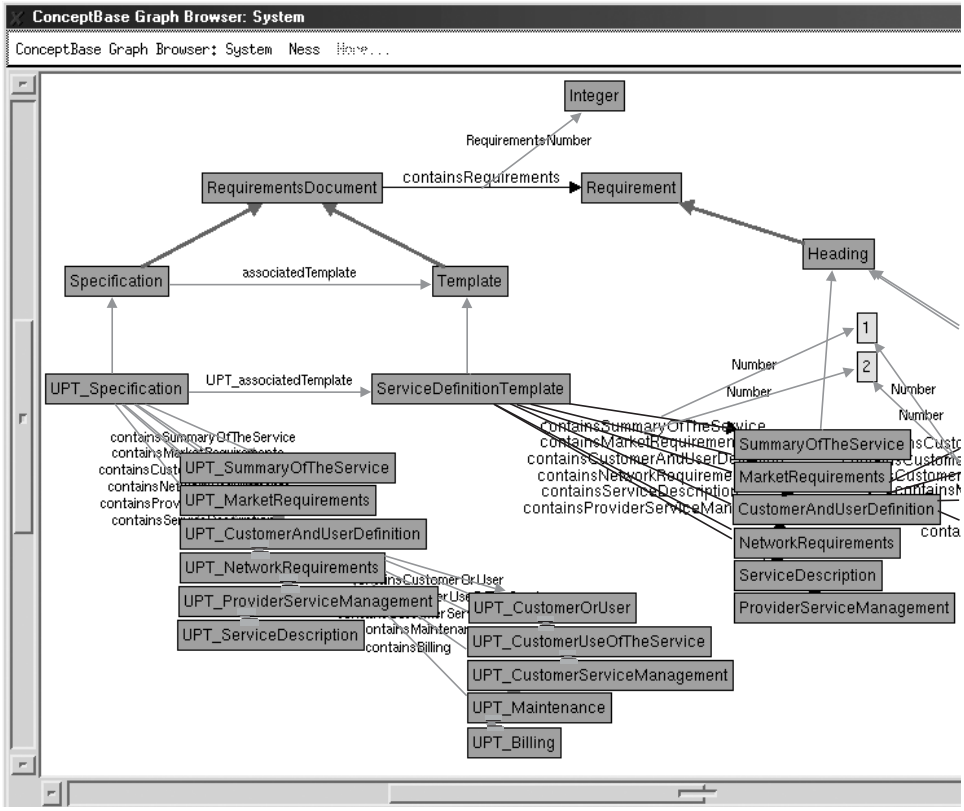


Figure 4.14
Specifications and their templates. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

4.6.2.2.2 Requirements Some parts of the `Requirement` subclass hierarchy are shown in figure 4.15. Each individual concept is implemented as a Telos class and is connected via attributes to the other classes with which it has relationships. Most subclasses have some subclass-specific attributes, but all subclasses also inherit all the attributes of their superclasses.

Some of the *general requirements attributes*, which each requirements subclass inherits, are shown in figure 4.16. The meaning and usage of these attributes as follows:

- Requirements ID: Each requirement has a unique identifier, which is the name under which the requirement is stored in ConceptBase.
- Requirements Number: Requirements contained within a requirements structure are hierarchically numbered using appropriate number levels (e.g., 4.6.2.10).

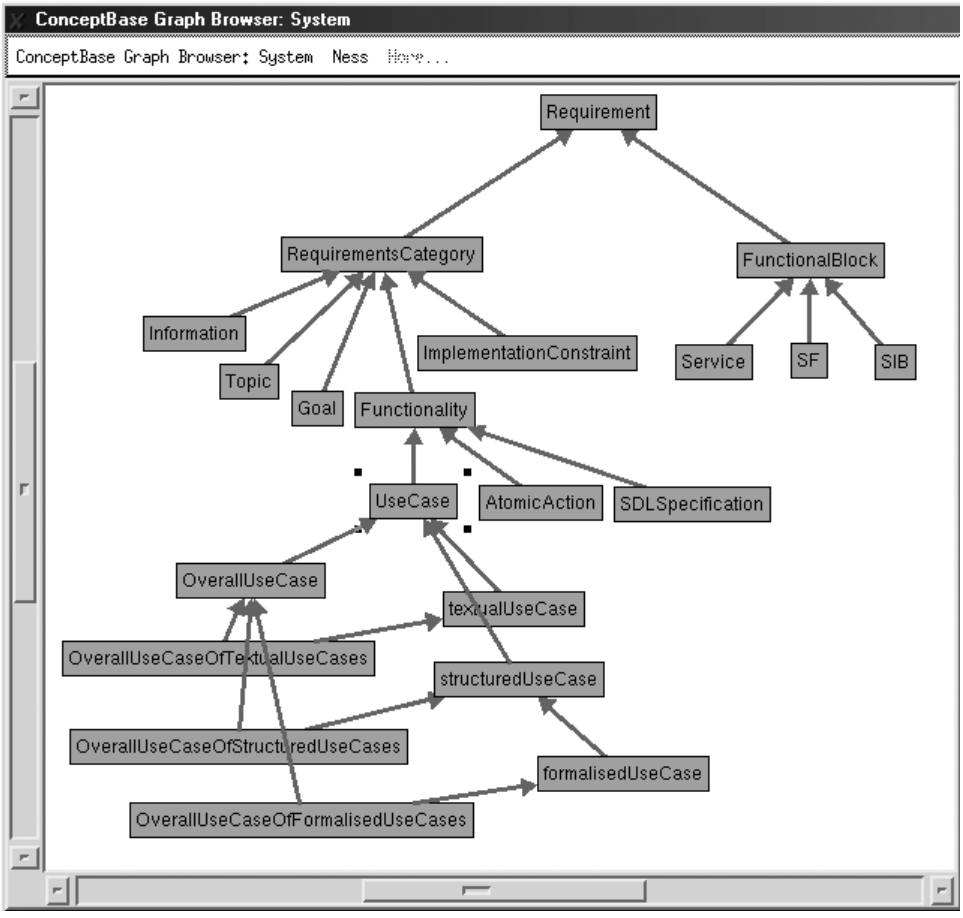


Figure 4.15
Excerpt from the `Requirement` subclass hierarchy. Attribute link ($\xrightarrow{\text{label}}$), subclass link (⟶), instance link (⟶), Telos object (▭)

• Date and Time of Creation: This is the date and time at which the requirement was created.

• Date and Time of Last Modification: This is the date and time at which the requirement was last modified.

• by Agent: This attribute of the structure link stores the name of the requirements engineer who inserted the requirement into the RATS tool.

• Source: This attribute names the stakeholder who initially stated the requirement and wanted it to be considered.

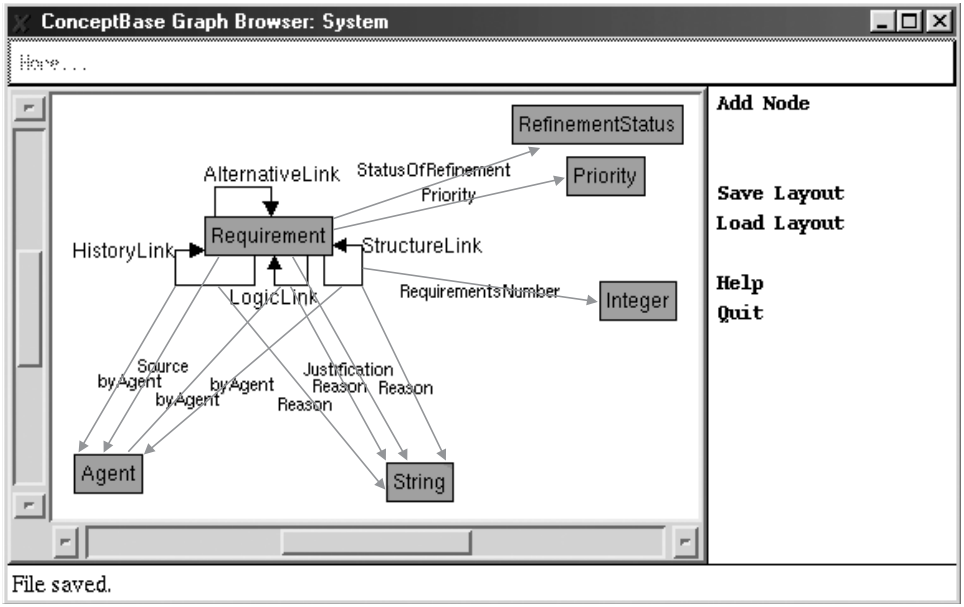


Figure 4.16
Excerpt from general requirements attributes. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ( )

• Justification: Here the rationale and goals behind the requirement are outlined. This attribute is crucial for requirements traceability, especially when the existence of the requirement has to be justified at a later point.

• Priority: Each requirement must have a priority assigned to it. There are currently three levels of priority, depending on the importance of the requirement:

- `Mandatory`: This requirement is absolutely essential.
- `Desirable`: This requirement should be implemented.
- `Optional`: It would be good to fulfill this requirement.

• Status of Refinement: This attribute states to what extent the requirement has been developed. Depending on the requirements subclass, this attribute can have different meanings. It shows the extent to which

- a topic has been addressed;
- information has been considered;
- a goal has been satisfied;
- functionality has been achieved;
- an implementation constraint has been considered.

There are three values possible for this attribute:

▪ `Not Refined`: The requirement has only been stated; it has not yet been further developed.
▪ `Partially Refined`: The refinement process of this requirement has started; however, further development is necessary.
▪ `Completely Refined`: This requirement has been developed to the point that it is fully satisfied by successive requirements.

• Status of Agreement: Inclusion of a requirement can make sense only if all the involved parties agree to it, as well as to its specification.

▪ `Agreed`: Stakeholders unanimously consent to the requirement and its specification.
▪ `Pending`: Some stakeholders agree to the definition of the requirement, some do not. This disagreement has to be settled by negotiation and by editing the requirement.
▪ `Rejected`: All parties have agreed that this requirement should be deleted.

• Agreed By: This attribute lists all those stakeholders who have given their consent to the requirement.

• Rejected By: This attribute lists all those stakeholders who have not given their consent to the requirement.

There are two more requirements subclasses with additional attributes that are worth mentioning:

• `NewRequirement`: Any requirement automatically becomes an instance of the class `NewRequirement` if there is no history link pointing to it. In such a case an additional attribute must be specified, giving the justification for the new requirement (see figure 4.17).

• `RejectedRequirement`: Any requirement automatically becomes an instance of the class `RejectedRequirement` if it has `Rejected` as its agreement status. In this case an additional attribute must be specified, giving the reason for the rejection of the requirement (see figure 4.17).

There are two rules that ensure any requirement that satisfies either of the conditions just mentioned becomes an instance of the appropriate requirements subclass.

```
Individual RequirementRule in Class with
  attribute,rule
    RejectedRequirement_rule : $ forall f1/Requirement (f1
    StatusOfAgreement Rejected) ==> (f1 in RejectedRequirement) $;
    NewRequirement_rule : $ forall f1/Requirement not(exists e1/
    Requirement (e1 HistoryLink f1)) ==> (f1 in NewRequirement) $
  attribute,comment
```

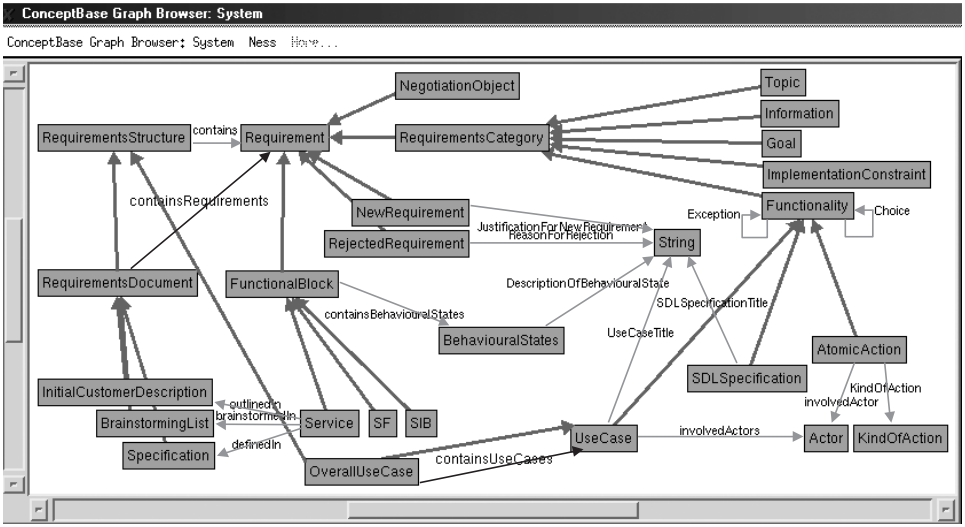


Figure 4.17
An excerpt from subclass-specific requirements attributes. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

```
    comment2 : "The RejectedRequirement_rule automatically makes
    each requirement which has the status Rejected into an
    instance of class RejectedRequirement.";
    comment7 : "The NewRequirement_rule automatically makes each
    requirement which has no History-link pointing to it into an
    instance of class NewRequirement."
end
```

• *Nonfunctional requirements* Nonfunctional requirements (i.e., information, topics, goals, and implementation constraints) have no additional subclass-specific attributes. These classes have been defined only to provide subclass-specific guidance and to improve performance by restricting the search space for Telos queries. The use of the general requirements attributes for nonfunctional requirements is therefore the same as for functional requirements and hence is covered by the following item in the list.

• *Functional requirements* All functional requirements belong to (i.e., are subclasses of) the class `Functionality`. As can be seen in figure 4.18, this class has two attributes (`Choice` and `Exception`) to link a functional requirement with other requirements specifying parallel and exceptional behavior. These two attributes are instances of the meta-attribute `necessaryInc` and thus must be specified for each functional requirement at some point during the development life cycle.

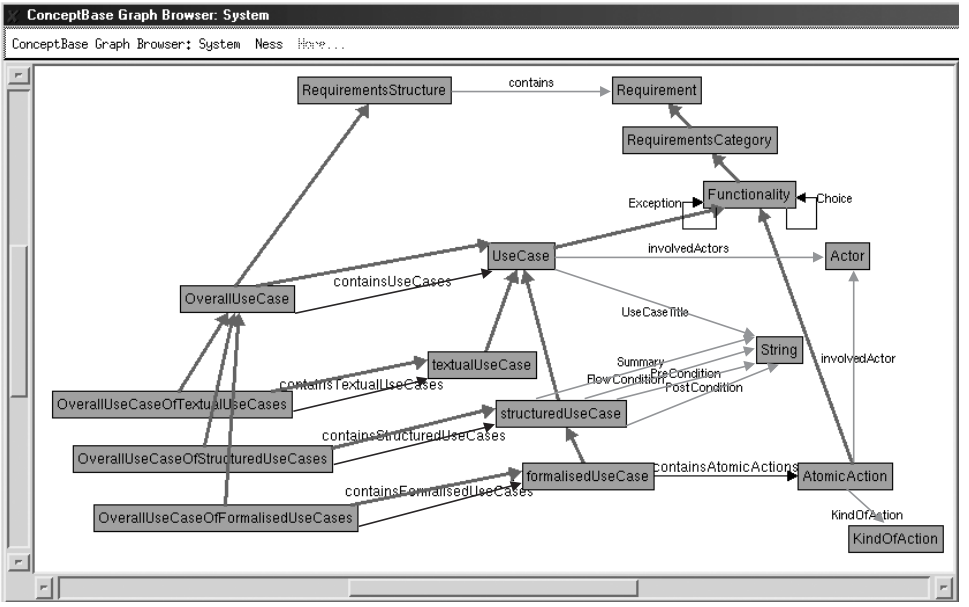


Figure 4.18
Another excerpt from subclass-specific requirements attributes. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

Functional requirements are decomposed using a three-stage use case design process. The Telos model containing the use case subclass hierarchy is shown in figure 4.18. All the different kinds of use cases are defined as subclasses of the class `UseCase` and thus inherit the attributes, rules, and constraints of this class. Specifying the different kinds of overall use cases as subclasses of the use cases with which they are associated (e.g., `OverallUseCaseOfStructuredUseCases` is a subclass of `structuredUseCase`) allows the recursive inclusion of use cases, as well as of overall use cases, hierarchically in high-level overall use cases.

In order to illustrate the inheritance of attributes, the class `OverallUseCaseOfStructuredUseCases` is employed as an example. Table 4.3 shows the implementation of the attributes of `OverallUseCaseOfStructuredUseCases`. Names of attributes are shown as they are displayed to the tool user.

4.6.2.2.3 Information Retrieval In addition to displaying and enabling browsing through requirements and the contents of requirements structures in a straightforward way, a knowledge-based development tool needs to provide a set of powerful query facilities that are able to retrieve specific information from the knowledge base. Some queries employed by the RATS tool are hidden from the user, as they

Table 4.3
Assignment of `OverallUseCaseOfStructuredUseCases` attributes to class attributes

Attribute of `OverallUseCaseOfStructuredUseCases`	Class attribute
Requirements ID	Name of the Telos object to be defined
Requirements No	`Requirement!RequirementsNo`
Date and time of creation	`RequirementsObject!CreationDateAndTime`
Date and time of last modification	`RequirementsObject!LastModificationDateAndTime`
Created by	`Requirement!HistoryLink!byAgent`
Source of requirement	`Requirement!Source`
Reason for requirement	`Requirement!HistoryLink!Reason`
Priority	`Requirement!Priority`
Status of refinement	`Requirement!StatusOfRefinement`
Status of agreement	`RequirementsObject!StatusOfAgreement`
Agreed by	`RequirementsObject!StatusOfAgreement!AgreedBy`
Rejected by	`RequirementsObject!StatusOfAgreement!RejectedBy`
Use case title	`UseCase!UseCaseTitle`
Involved actors	`UseCase!involvedActors`
Use case summary	`structuredUseCase!Summary`
Precondition	`structuredUseCase!PreCondition`
Flowcondition	`structuredUseCase!FlowCondition`
Postcondition	`structuredUseCase!PostCondition`

are automatically executed by the client logic of the RATS client in order to display relevant information to the user. Other queries, however, have to be issued directly by the user. These are mostly queries related to requirements management, such as the history of a requirement or the impact of a change to a requirement. To give an example, a query about the impact of a requirements change can be implemented in the frame syntax of the Telos language as follows:

```
Individual ChangeOfRequirementAffectsLogically in
GenericQueryClass isA Requirement with
  attribute,parameter
    R : Requirement
  attribute,constraint
    con : $ (~R LogicLink this) or (exists e1/Requirement ((e1 in
    ChangeOfRequirementAffectsLogically[~R/R]) and (e1 LogicLink
    this)))$
end
```

As can be seen in its constraint definition, the query calls itself (i.e., it is a recursive query). This demands that the query be an instance of the system class `Magic`. Addi-

tionally, since the query allows the user to ask for the impact of a change in *any* requirement, a parameter `R` is defined that is used to specify the requirement that is to be changed. This makes the query generic; it is thus necessary to define the query as an instance of `GenericQueryClass`. The advantage of the definition provided here is that despite its being recursive, loops of logical links do not cause the query to crash. However, the query can be very slow to return an answer when a considerable number of requirements have been defined.

4.6.2.2.4 Intermodel Consistency The RATS tool has in its domain layer many domain models (see figure 4.4) that contain domain knowledge conceptually expressed in Telos. An attempt has been made to create these domain models as independently of one another as possible. This has great advantages during modeling, as it separates the various domain areas and allows the use of different kinds of domain models. However, the independence of the models results in the danger of inconsistent use of their contents, raising the topic of *intermodel consistency*.

The RATS tool makes use of rules and constraints to guarantee meaningful usage of the libraries contained in the domain models, taking care that elements taken from different domain libraries fit together. There are several ways in which this is achieved. Using passive guidance, permanent as well as temporary user-defined constraints can be employed. However, in order to provide active guidance, intelligence rules and intelligence objects are necessary. The following discussion illustrates the latter approach. It shows a Telos rule that is part of a pool of object-related intelligence rules. The rule ensures that service definitions specify only basic network services that are also offered by at least one of the networks in the service being specified. In cases in which the basic service specified is not provided by any network, the tool user is informed with the help of the object-related intelligence object `Int25`.

In order to define the intelligence rule, we first have to specify another rule that makes sure that a link is created between the network and the services it offers:

```
Individual NWRule in Class with
  rule,attribute
    offers_rule : $forall f1/NW e1/Library
      (exists e2/BasicNWService (f1 offers e2) and (e1 in e2)) ==>
      (f1 offers e1)$
end
```

Now we can specify the intelligence rule:

```
NetworkAndBasicServicesFitTogether_rule : $forall f1/Specification
  (exists e1/NW e2/Library
    (exists e3/NegotiationObject (f1 contains e3) and (e3
    linksToLibrary e1)) and
```

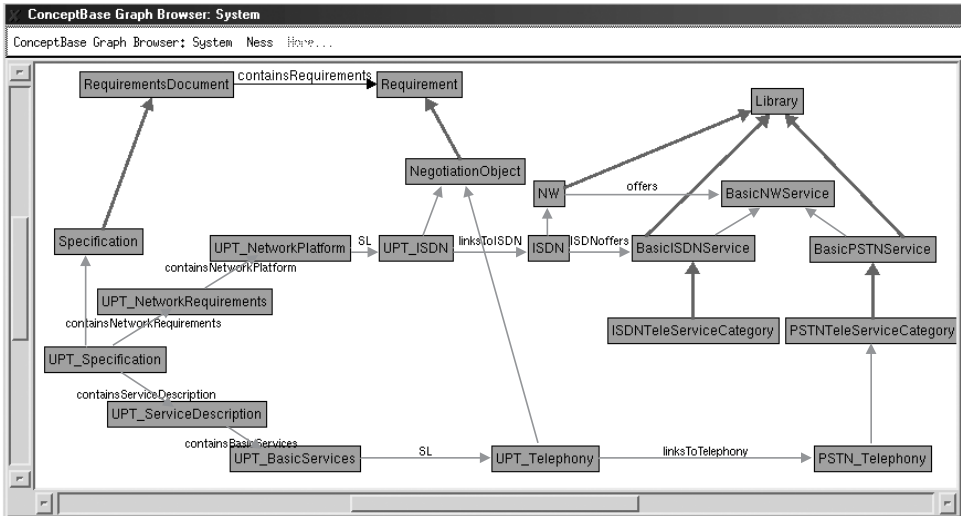


Figure 4.19
Example of intermodel inconsistency. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\Longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

```
    (exists e5/NegotiationObject (f1 contains e5) and (e5
    linksToLibrary e2)) and not(e1 == e2) and not(e1 offers e2))
==> (f1 IntelligenceObject Int25)$
```

The functionality of this rule is illustrated in figure 4.19, again using the example of the UPT service. To make the example more intuitive, meaningful names have been selected rather than the names created automatically by the RATS client. The values of the different variables (`f1`, `e1`, `e2`, etc.) in this particular example are shown in the figure. Using these values, the rule states that if the UPT specification contains at least one network and at least one basic service, then all the specified basic network services need to be offered by at least one of the specified networks; if this is not the case, the intelligence object `Int25` is assigned to the `UPT_Specification`. In the example of figure 4.19, the dashed attribute `ISDN!offers` is not generated by the `offers_rule`; that is, the network and the basic services included in the UPT specification do not fit together, and the `NetworkAndBasicServicesFitTogether_rule` links `Int25` to the `UPT_Specification`.

The way intelligence rules are defined is of crucial importance. For instance, the rule here is not triggered when the specification is still incomplete. If any of the five links (*l1*, *l2*, *l3*, *l4*, or *l5*) has not yet been specified, the UPT service definition is incomplete, but not inconsistent; thus the rule is not supposed to be triggered. Such a case is considered in the preceding definition of the rule.

The preceding description makes clear that intermodel consistency is ensured by the intelligence models; however, as can be seen in figure 4.19, it also involves parts of the negotiation models (e.g., the object `NegotiationObject`). Nevertheless, since active help for the resolution of inconsistency was one of the aims in the development of the RATS tool, intermodel consistency has been implemented as part of the intelligence models rather than of the negotiation models.

4.6.3 The Negotiation Models

The negotiation models perform several tasks. Their main task is to provide an interface between the development models and the domain models. This interface allows the use of different kinds of domain models. Furthermore, as a result of this interface, active, domain-specific guidance can be offered, together with the ability to reuse previously defined domain objects and requirements specifications. Intermodel consistency could in theory be another task of this layer; however, RATS provides this by means of the intelligence models. Currently, much of the potential functionality of the negotiation models belongs to the future development of the RATS tool.

4.6.4 The Domain Models of the Domain Layer

The RATS domain models are at various stages of implementation. The most complete one is the model containing information on network transport capability. The other models are not yet or only partially implemented. The following sections therefore give only some introductory information on each domain model. Implementational details are given only for the model containing the network transport capability.

- *Customer profile* The customer profile model contains information about each customer. For this purpose it includes templates for relevant customer information. The database holds information about the customers themselves, the subscribed networks, the subscribed bearer, tele- and supplementary services, the interface structure, the CPE, and the local exchange to which the customers are connected. Depending on the kind of customer (i.e., private or business) an appropriate template is used. Figure 4.20 shows an abridged version of a possible customer profile template for a private customer.
- *Customer premises equipment* The CPE model contains the characteristics of the various kinds of equipment that can be installed at the end of the network on the customer's premises. This can be a private automatic branch exchange (PABX), an ISDN or analog telephone, a local area network (LAN), etc. Since CPE is directly connected to a network, the CPE and network transport capability models are closely linked. The CPE model contains information about the features, abilities, and characteristics of the various pieces of CPE (e.g., display, camera, encryption

name (surname, initials, title)	personal profile: e.g.,
address	working life
date of birth	house type
sex	current bill and status
language spoken	credit worthiness
language written	credit status
telephone number	credit limit
account code number	QoS
current status, e.g., fault reported	installation
current product profile	network
current product configuration	internal wiring
pricing and possible discount structures	CPE
service level agreements:	preferred billing media
type	preferred payment method
response times	deposit

Figure 4.20
Example of a customer profile template for a private customer

facility, redial function), as well as their connection to the network (e.g., access interfaces, connection points). CPEs suitable for use with ISDN can also be classified according to their ISDN reference points (R-, S-, T-interface), as well as their functional groups (Terminal Equipment [TE] 1/2, Terminal Adapter [TA], Network Terminator [NT] 1/2), according to ITU-T 1993c. Additionally, the CPE's interface structures (e.g., 2B+D16) (ITU-T 1993d) are included in the description.

• *Network transport capability* The network transport capability model contains the capabilities of the various telecommunications networks and is one of the major domain models. Each network (e.g., PSTN, ISDN) for which services are to be developed with the assistance of the RATS tool needs to be conceptually modeled in the network transport capability model using Telos. The current version of RATS contains a very comprehensive model of the ISDN network. Relevant information has been collected from many sources, such as standards (e.g., I.xxx [ITU-T 1988–2000] and ETS 300 xxx [ETSI 1990–1997]) and books (e.g., Bellamy 1991; Kessler 1998; PTT Telecom 1993), as well as various ISDN Internet pages (e.g., Kegel 1996). The present version of the ISDN model includes the following information:

- generic network characteristics (e.g., 64 kbps)
- combinations of network characteristics (e.g., 64 kbps, unrestricted, 8 kHz, structured) recommended by international bodies (e.g., ITU-T, European Telecommunication Standards Index [ETSI])
- basic ISDN services (bearer services and teleservices) characterized by service attributes (see figure 4.21)

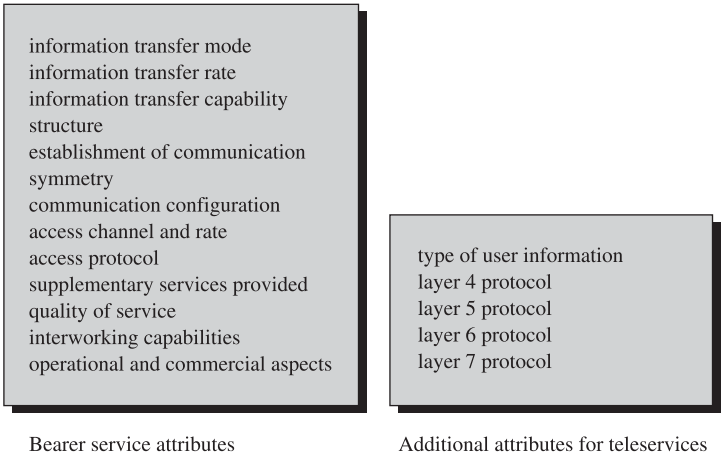


Figure 4.21
ISDN service attributes for basic ISDN services (see ITU-T 1993b)

- information about service availability
- interoperability among basic services
- user-network interfaces and their attributes
- textual descriptions of the domain objects
- references to the appropriate standards

Figure 4.22 shows a small part of the comprehensive ISDN domain model. It illustrates how conceptual domain models can be used to teach newcomers to the domain about certain topics. The extract shows that a telecommunications network offers basic network services, which in turn are characterized by service attributes that can be assigned to attribute categories. If more information about service attributes is required, the user can display the instances of the service attributes and their categories. The model also contains information on the relevant standards. In the example shown in figure 4.22, ITU-T I.140 (1993a) is the appropriate standard. Each network covered by the model can be accessed via a user-network interface. The basic network services can be broken down into two categories: bearer services and teleservices. In the case of ISDN, ten bearer service categories and eight teleservice categories have been defined. If one of these services is selected, its attributes can be displayed, and more information on each of the service categories could be retrieved.

- *Supplementary services* The supplementary services domain model contains information about supplementary services, their characteristics, and their specifications. The current version of the RATS methodology and its implementation in the RATS tool are oriented toward the Intelligent Network concept. This model therefore

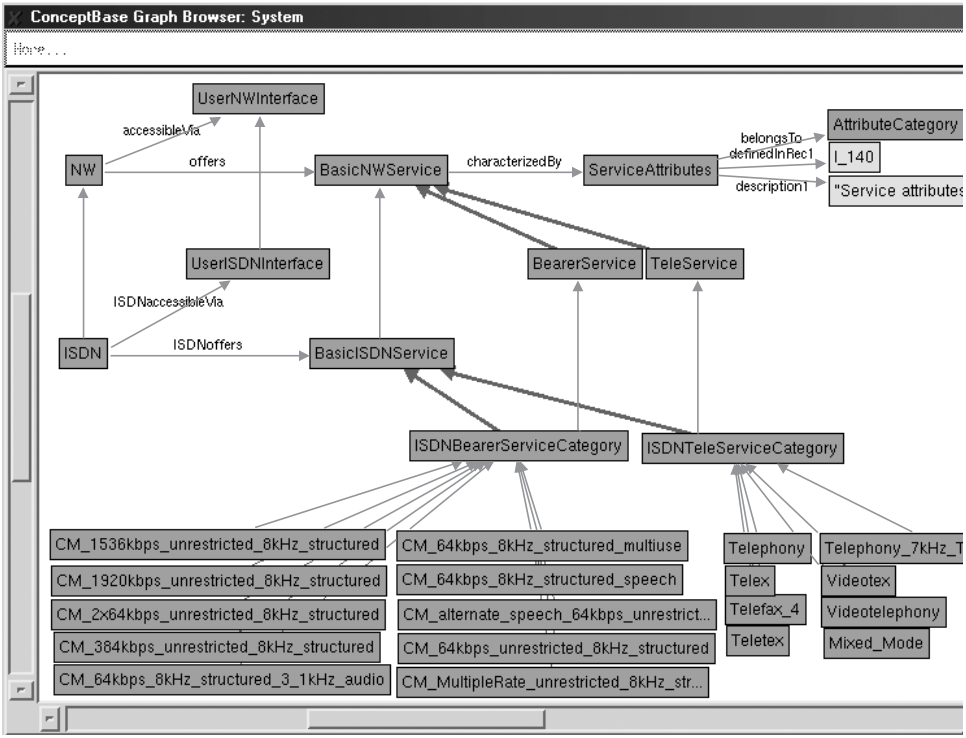


Figure 4.22
The ISDN domain model (partial). Attribute link ($\xrightarrow{\text{label}}$), subclass link (⟹), instance link (⟶), Telos object (▭)

contains significant parts of the Intelligent Network architecture of the Q.12xx series (ITU-T 1992–2000).

• *Feature interaction* The feature interaction domain model belongs to future work on the RATS implementation. Different levels of sophistication could be introduced. A simple approach would be to create and store a conceptual model of known feature interactions in the knowledge base. A more sophisticated approach could assign characteristics to services and service features that are relevant for feature interaction detection. If several features were combined during service creation, their characteristics could point out unknown feature interactions.

• *Network interoperation* Since the current telecommunications network consists of a wide variety of architectures, interoperation among different network types is essential. This model will in future contain information about interoperability issues. Such information can be found in, for example, ITU-T 1995 and 1993b, and addresses conversion functions like analog-to-digital and digital-to-analog conversion,

conversion of signaling systems, and protocols. Such conversions can become necessary within a local exchange, at a transit exchange, or at international gateway offices.

• *Switch* Switches are becoming more and more sophisticated in today's telecommunications systems. The demand for more services to be offered to the customer is resulting in a drastic increase of functionality contained in the switch. Many service features are better provided by the switch rather than the IN, in order to reduce the signaling overhead needed for IN services (Thörner 1994). The switch domain model is also part of the future work on the project. It will basically contain a table listing available switch types and the functionality they offer (e.g., bearer services, teleservices, switch-based supplementary services, and operator services).

• *Data dictionary* The data dictionary also belongs to the future work on the RATS tool. Its task will be to store definitions used during service specification and to establish a common vocabulary. Agreement on certain definitions of terms will help to reduce ambiguity and will contribute to formality. Recommendations like I.11x (ITU-T 1993–1997) and Q.1290 (ITU-T 1998) contain essential definitions that can be stored in the data dictionary.

• *Expanding the domain layer* The domain layer has great potential for expansion. Additional domain models or extensions to the current models could drastically increase the help that the RATS tool can offer to the service developer.

Parts of the domain models can be extended relatively easily when the RATS tool is being used. The development of new services will result in the specification of new service features that, when fully specified and tested, can be included in the domain model of the supplementary services and be reused in future developments.

4.7 Model Analysis

Complex models require tremendous effort to ensure correctness. It is easy to generate a model that looks impressive, but it is a great challenge to ensure that it is correct. Two aspects of ensuring model correctness need to be addressed:

• *Verification* ensures that the model is consistent and correct in itself.

• *Validation* ensures that the model resembles the real world and correctly represents the aspects of the world it intends to model.

A combination of several approaches have been used to check that the RATS models are correct:

• *Reviews* The most common approach to model checking is careful analysis of them and comparison with the real world. Such analysis and comparison requires

time, detailed domain knowledge, and a clear mind. It is beneficial if several people can check the models. Analysis and comparison of this type is a static approach to model analysis and is good for verification as well as validation.

- *Testing* Testing is very important for checking constraints, rules, and queries. Situations should be created within the model that cause constraints and rules to be triggered at the appropriate time. In the case of RATS, testing is used in two ways:

- After the definition of constraints, rules, and queries, the effect of inserting and deleting objects is checked.

- In the case of constraints, attempts are made to insert objects that are known to violate a particular constraint (i.e., the constraint needs to prevent the insertion). For instance, the following constraint can be checked by trying to insert an ISDN bearer service variant with the attribute `Available` even though its bearer service category is not available.

```
Individual ISDNBearerServiceVariantConstraint in Class with
  constraint,attribute
    Provision_Con : $ forall f1/ISDNBearerServiceVariant
      (f1 hasBearerServiceVariantProvision Available)
  ==> (exists e1/ISDNBearerServiceCategory (f1 isA e1) and
      (e1 hasBearerServiceCategoryProvision Available))$
end
```

This constraint will prevent the insertion of the ISDN bearer service variant.

- In the case of rules, different conditions are created that show that the rule is triggered in the appropriate situations. With the help of queries, the correct functioning is checked:

```
Individual RequirementRule in Class with
  rule,attribute
    RejectedRequirement_rule : $ forall f1/Requirement
    (f1 StatusOfAgreement Rejected)
    ==> (f1 in RejectedRequirement)$
end
```

In this case, two requirements are inserted into ConceptBase: one with the agreement status `Rejected` and another with the agreement status `Accepted`. A query class is then defined that retrieves all those requirements that are instances of the class `RejectedRequirement`. The first requirement should be returned by the query; the second one should not. If this is what actually happens, there is a reasonable level of confidence that the rule works correctly.

- In the case of queries, different objects are created that show the boundary of the subclass defined by the query. For instance, if all requirements are to be found that

have not yet been decided on (i.e., they do not have an agreement status of either `Accepted` or `Rejected`) then the following query is defined:

```
Individual PendingRequirements in QueryClass isA Requirement with
  attribute,constraint
    rule : $ not (this StatusOfAgreement Agreed) and not (this
    StatusOfAgreement Rejected) $
end
```

The correctness of the query can be checked by inserting some requirements into the knowledge base that are `Agreed`, some that are `Rejected`, and some without any agreement status defined. Only the last type will be returned by the query if it works correctly.

• Constraints also can be used to ensure that the contents of the knowledge base are consistent at higher levels of abstractions. This is where the intelligence of the RATS tool comes in. For instance, the RATS server includes a constraint that prevents a functional requirement from being refined into a goal (i.e., the model is consistent with the development methodology).

It has to be stressed, however, that it can be a great challenge to verify the correctness of complicated conceptual models and the constraints, rules, and queries contained in them.

4.8 Example Models of the Application

The previous sections have given several examples of different aspects of the RATS tool. This section illustrates parts of the interaction between the RATS client and the RATS server. The examples presented in the section partially show how telecommunications service development can take place using RATS.

Assume that a new telecommunications service (the UPT service) is to be specified. When starting this new project (i.e., UPT), the RATS client creates a new initial customer description, a brainstorming list, and a specification document. These three UPT-specific documents are created as instances of their respective classes (see figure 4.23). (Note that while the RATS client generates unique names for all objects based on the current system time of the server, in the example provided here, we use more intuitive names for the documents to improve readability).

```
Individual UPT in Project with
  attribute,icd
    i : UPT_InitialCustomerDescription
  attribute,bl
    b : UPT_BrainstormingList
```

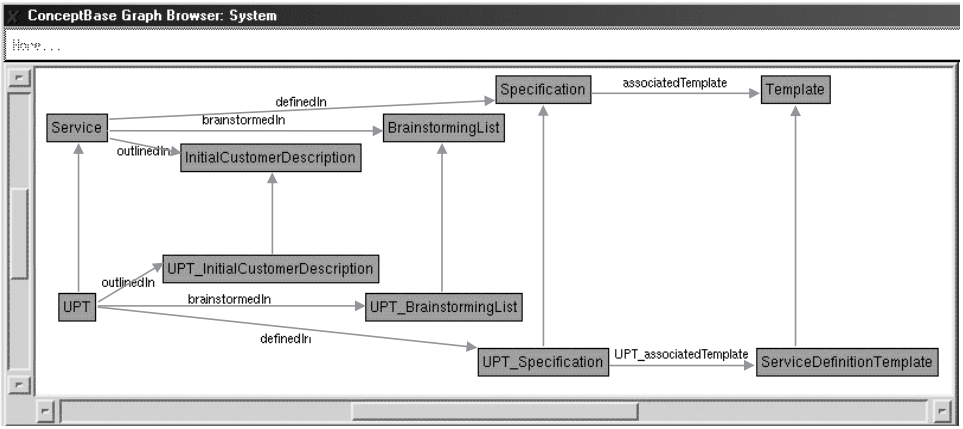


Figure 4.23
Setup of a new service development project. Attribute link ($\xrightarrow{\text{label}}$), subclass link ($\longrightarrow$), instance link ($\longrightarrow$), Telos object ($\square$)

```
  spec,attribute
    s : UPT_Specification
end
```

In order to allow changes to the document templates, a class `Template` has been defined. This permits the definition of various templates customized for specific projects or customers. The service definition template contains headings and subheadings that have to be contained in the specification. This means that `UPT_Specification` is an exact copy of `ServiceDefinitionTemplate`. However, all refinement of requirements takes place in the object `UPT_Specification`. The original `ServiceDefinitionTemplate` is not affected by service development.

Once all the documents are initially set up, the first task to be undertaken is the completion of the initial customer description. The customer writes only an outline of the service to be developed, which will be stored as an attribute of `UPT_InitialCustomerDesription`:

```
Individual UPT_InitialCustomerDescription in
InitialCustomerDescription with
  CreationDateAndTime,attribute
    cdat : "Fri Jul 28 14:52:57 MDT 2000"
  AffiliationOfCustomer,attribute
    aoc : "University of Calgary"
  CreatedBy,attribute
    cb : Armin Eberlein
```

```
  PlaceOfCreation,attribute
    cp : "Calgary"
  StatusOfAgreement,attribute
    soa : Agreed
  Brainstorming,attribute
    bl : UPT_BrainstormingList
  LastModificationDateAndTime,attribute
    lmdat : "Fri Jul 28 14:53:16 MDT 2000"
  attribute,containsRequirements
    cr1 : f961527509378
end
```

The customer's description is included as a string in the requirement `f961527509378`:

```
Individual f961527509378in Requirement,Goal with
  Source,attribute
    source : Customer
  Priority,attribute
    priority : Mandatory
  StatusOfAgreement,attribute
    status : Agreed
  Description,attribute
    Description : "I want to be able to make a phone call from any
    phone and to receive phone calls at any phone with one unique
    number."
end
```

After completion of the initial customer description, requirements are added to the brainstorming list as they are generated during the brainstorming session. These requirements are usually directly defined as attributes of the brainstorming list:

```
Individual UPT_BrainstormingList in BrainstormingList with
  CreationDateAndTime,attribute
    cdat : "Fri Jul 28 14:52:57 MDT 2000"
  LastModificationDateAndTime,attribute
    lmdat : "Fri Jul 28 14:53:16 MDT 2000"
  Action,attribute
    meet1 : f964817593362
  StatusOfAgreement,attribute
    soa : Agreed
  Specification,attribute
```

```
    spec : UPT_Specification
  attribute,containsRequirements
    cr1 : f961312039802;
    cr2 : f961392879171;
    cr3 : f964189089801;
    cr4 : f964817123890;
    cr5 : f964812983012;
    cr6 : f960982312397;
    cr7 : f964782673899
end
```

Next the challenging task of specifying the telecommunications service using the service definition template must be completed. This template contains some headings that serve as reminders that certain aspects have to be addressed. The requirements engineer has several possible requirements operations (see table 4.4) that can be used to work on the specification. Each operation has various effects on the attributes of the affected requirements. Some of the changes in the links can be seen in table 4.4.

```
Individual UPT_Specification in Specification with
  CreationDateAndTime,attribute
    cdat : "Fri Jul 28 11:52:57 MDT 2000"
  LastModificationDateAndTime,attribute
    lmdat : "Fri Jul 28 15:53:16 MDT 2000"
  Action,attribute
    meet1 : f964817526789
  StatusOfAgreement,attribute
    soa : Agreed
  attribute,containsRequirements
    cr1 : UPT_SummaryOfTheService;
    cr2 : UPT_MarketRequirements;
    cr3 : UPT_CustomerAndUserDefinition;
    cr4 : UPT_ServiceDescription;
    cr5 : UPT_NetworkRequirements;
    cr6 : UPT_ProviderServiceManagement
  attribute,description
    description1 : "The specification addresses a list of topics
    necessary to define a new service. It should be done in
    connection with the customer. It is still informal and
    contains only written text. The requirements can be stepwise
    refined and the final version of the specification is supposed
    to contain a reasonably complete picture of the new service."
end
```

Table 4.4
Possible operations on requirements in RATS

		Old Requirement		New Requirement		Change of Links		
Operation	Description	in KB	in doc	in KB	in doc	History	Logic	Structure
create	Create a new requirement	n/a	n/a	yes	op	n/a	op	op
edit	Minor textual editing without recording of the change	no	no, n/a[a]	yes	yes, op[a]	no	no	no, op[a]
refine	A parent requirement is refined into subrequirements	yes	yes, n/a[a]	yes	op	yes	yes	op
collapse	Several similar requirements are simplified into one requirement	yes	no, n/a[a]	yes	yes, op[a]	yes	yes	yes, op[a]
replace[b]	A requirement is replaced by one or more other requirements	yes	no, n/a[a]	yes	yes, op[a]	yes	yes	yes, op[a]
delete	A requirement has been rejected	yes	no, n/a[a]	n/a	n/a	no	no	yes, no[a]
cut & paste	The location of requirements in a requirements structure is changed	n/a	n/a	n/a	n/a	no	yes	yes
create alternative	Alternative ideas can be pursued	yes	yes, n/a[a]	yes	op	yes	yes	op
create parallel behavior	Parallel behavior is being defined	yes	yes, n/a[a]	yes	op	yes	yes	op
create exceptional behavior	Exceptional behavior is being defined	yes	yes, n/a[a]	yes	op	yes	yes	op

Note: KB: knowledge base; doc: document; n/a: not applicable; op: optional.
a. Depending on the original requirement.
b. Similar to edit, however, the change is recorded.

The following example illustrates the interaction between the RATS client and RATS server. In order to keep the example short, the simple *edit* operation is used, and it is assumed that no errors occur during the execution of the procedure. This *edit* operation is applied for minor editorial changes, in this example, a spelling mistake is to be corrected.

The existing requirement, with the requirements ID `f974817577859`, expressed in the Telos frame syntax, is

```
Individual f974817577859 in Requirement with
  Description,attribute
    Description: "General secrty requirements are necessary in
    order to prevent several ways of misuse: Fraudulent use,
    eavesdropping of information exchanged, eavesdropping of UPT's
    user profile, disclosure of user's physical location."
  HistoryLink,LogicLink,attribute
    refinedInto1: f973948202984
  LastModificationDateAndTime,attribute
    Lmdat: "Thu Jul 27 11:40:07 MDT 2000"
end
```

The RATS tool user wants to correct the spelling mistake (i.e., wants to change the word "secrty" to "security"). In order to do that, he or she loads the requirement with the requirements ID `f974817577859`, and the GUI displays this requirement, together with its attributes. The textual description of the requirement is shown in an editable window, in which the user corrects the spelling mistake and presses the EDIT button that is part of the GUI in order to update the knowledge base (KB). The following sequence of actions will then take place:

1. The GUI passes the operation to be performed (i.e., *edit*), the requirements ID (i.e., `f974817577859`), and the attribute to be edited (i.e., `Description`) to the client logic.

2. The client logic passes the requirements ID (i.e., `f974817577859`) and the attribute to be edited (i.e., `Description`) to the frame generator.

3. The CL triggers the FG to *ask* for the present version of the requirement.

4. The CL asks the FG to generate a frame containing only those attributes that have to be deleted by removing all the attributes and classifications that do not change:

```
f974817577859 with
  Description,attribute
    Description: "General secrty requirements are necessary in
    order to prevent several ways of misuse: Fraudulent use,
```

```
    eavesdropping of information exchanged, eavesdropping of UPT's
    user profile, disclosure of user's physical location."
  LastModificationDateAndTime,attribute
    Lmdat: "Thu Jul 27 11:40:07 MDT 2000"
end
```

5. The CL asks the FG to *untell* the generated Telos frame from the KB.

6. The CL retrieves the present date and time from the UNIX system.

7. The CL passes the present date and time as parameters to the FG.

8. The CL asks the FG to generate a Telos frame by replacing the old `Description` attribute and `LastModificationDateAndTime` attribute with the new attributes:

```
f974817577859 with
  Description,attribute
    Description: "General security requirements are necessary in
    order to prevent several ways of misuse: Fraudulent use,
    eavesdropping of information exchanged, eavesdropping of UPT's
    user profile, disclosure of user's physical location."
  LastModificationDateAndTime,attribute
    Lmdat: "Fri Jul 28 08:31:47 MDT 2000"
end
```

9. The CL triggers the FG to *tell* the corrected Telos frame to the KB.

10. The CL triggers the GUI to notify the user of the outcome of the operation.

11. The CL triggers the FG to *ask* the KB for the complete new requirement with the ID `f974817577859`, as well as its attributes:

```
Individual f974817577859 in Requirement with
  Description,attribute
    Description: "General security requirements are necessary in
    order to prevent several ways of misuse: Fraudulent use,
    eavesdropping of information exchanged, eavesdropping of UPT's
    user profile, disclosure of user's physical location."
  HistoryLink,LogicLink,attribute
    refinedInto1: f973948202984
  LastModificationDateAndTime,attribute
    Lmdat: "Fri Jul 28 08:31:47 MDT 2000"
end
```

12. The CL triggers the GUI to display the corrected requirement to the RATS user.

This example illustrates how a simple user operation results in a complex sequence of transactions. More-complex operations and the implementation of error recovery procedures within the RATS client make its design a demanding challenge.

4.9 Critical Review of the Solution

The current implementation of RATS has two major disadvantages:

1. Lack of flexibility
2. Poor performance

The lack of flexibility is a problem inherent in all hierarchical conceptual models. Although hierarchies are excellent for structuring a model, having to change any higher levels of a hierarchical conceptual model is a major challenge. If one of the high-level concepts of the RATS methodology (such as `RequirementsObject`) were to change significantly, most of the subclasses and instances of this class would need to be adjusted. This would also require changes in the implementation of the RATS client.

Poor performance is the biggest problem with the RATS prototype, which currently runs on a SUN Enterprise Server 450. This configuration is far greater than the minimum requirements of the ConceptBase tool; however, the size of the RATS tool implementation demands such a configuration in order to run stably. Despite significant improvements in performance over previous ConceptBase versions, many concepts cannot be modeled because of the implementation's slow performance. This issue calls for a difficult compromise between neat, proper, and modular models using the full range of Telos features and acceptable performance of the prototype.

This research has suggested the following guidelines that should be remembered during modeling using RATS:

- Keep the models as small as possible.
- An application can consist of many classes (hundreds to thousands), but each class should only contain a few instances (Jarke, Jeusfeld, and Staudt 1999a).
- Use mainly "primary" rules; avoid chains of rules (i.e., rules that use information that has to be retrieved by other rules).
- Avoid recursion.
- Create subclass hierarchies to restrict the search spaces of queries.
- Keep queries as simple as possible; avoid calculated attributes.
- If possible, use temporary constraints rather than permanent constraints, as this avoids unnecessary database consistency checks.

These guidelines demand a trade-off between generic modeling (allowing a high degree of reusability) and performance. The more general class models are, the more instances they will contain, resulting in slower performance. Unfortunately, some of these guidelines are contrary to the nature of requirements engineering for large-scale projects. Such projects may result in thousands of requirements, all of which are

instances of the class `Requirement`. There might arise a need for the definition of even more subclasses of `Requirement` than there currently are in RATS, in order to reduce the search space in larger projects.

Good modeling practice tends to lead to slow response time. However, acceptable response time for an expert system is relatively small. Experience gained during this research shows that most queries need to be answered within 5 seconds to keep tool users satisfied, and maximum tolerable response time is 20–30 seconds. Unfortunately, it is easy to construct queries in RATS that take much longer to compute.

4.10 Conclusions

Developing complex software systems is difficult; artificial intelligence techniques can help meet this challenge. However, providing this type of support is nontrivial. The research presented in this chapter suggests that conceptual models are an effective way of providing support. The knowledge representation language Telos and its implementation in ConceptBase have proven to be very suitable for this task. The flexibility of the Telos language allows the modeling of complex domains in order to provide the necessary support.

The research presented in this chapter needs to be extended along two directions: First, the conceptual models used need to be refined to broaden the assistance provided to the telecommunications service developer. Second, the performance of ConceptBase needs to be improved to make it suitable for large-scale models and applications.

Notes

1. Here the term "methodology" is used rather than "method," since this is the more commonly used term in the telecommunications domain. Nevertheless, some software engineering authors define "methodology" as "the science of methods" (e.g., Graham 1994; Schach 1999), but others do not distinguish between the two terms and use them interchangeably (e.g., Pohl 1994). More details can be found in Gillies 1991.

2. The term "satisficed" is used in this context to indicate that NFRs are satisfied only within limits (Chung, Nixon, and Yu 1995).

3. `Ask`, `tell`, and `untell` are input-output commands of the ConceptBase server.

References

Balzer, R., N. Goldman, and D. Wile. 1978. "Informality in Program Specifications." *IEEE Transactions on Software Engineering* 4, no. 2: 94–103.

Bellamy, J. 1991. *Digital Telephony*. New York: Wiley.

Boehm, B. W. 1984. "Verifying and Validating Software Requirements and Design Specifications." *IEEE Software* 1, no. 1: 75–88.

Bouma, L. G., and H. Velthuijsen. 1994. *Feature Interactions in Telecommunications Systems*. Amsterdam: IOS Press.

Chung, L., B. A. Nixon, and E. Yu. 1995. "Using Non-functional Requirements to Systematically Support Change." In *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE'95)*, 132–139. Los Alamitos, CA: IEEE Computer Society.

ETSI (European Telecommunication Standards Index). 1990–1997. ETSI ETS 300 xxx Series of Standards: Integrated Services Digital Network. European Telecommunication Standards Institute, Cedex, France.

Gillies, A. C. 1991. *The Integration of Expert Systems into Mainstream Software*. London: Chapman and Hall.

Graham, I. 1994. *Object Oriented Methods*. Reading, MA: Addison-Wesley.

ITU-T (International Telecommunication Union). 1988–2000. ITU-T I.xxx Series of Recommendations: Integrated Services Digital Network. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1992–2000. ITU-T Q.12xx Series of Recommendations: Intelligent Networks. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1993–1997. ITU-T I.11x Series of Recommendations: Terminology. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1993a. ITU-T I.140 Recommendation: Attribute Technique for the Characterisation of Telecommunication Services Supported by an ISDN and Network Capabilities of an ISDN. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1993b. ITU-T I.210 Recommendation: Principles of Telecommunication Services Supported by an ISDN and the Means to Describe Them. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1993c. ITU-T I.411 Recommendation: ISDN User-Network Interfaces—Reference Configurations. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1993d. ITU-T I.412 Recommendation: ISDN User-Network Interfaces—Interface Structures and Access Capabilities. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1993e. ITU-T I.530 Recommendation: Network Interworking between an ISDN and a Public Switched Telephone Network (PSTN). International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1995. ITU-T Q.130x Series of Recommendations: Telecommunication Applications For Switches and Computers. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1998. ITU-T Q.1290 Recommendation: Glossary of Terms Used in the Definition of Intelligent Networks. International Telecommunication Union, Geneva, Switzerland.

ITU-T (International Telecommunication Union). 1999. ITU-T Z.100 Recommendation: Specification and Description Language (SDL). International Telecommunication Union, Geneva, Switzerland.

Jarke, M., M. Jeusfeld, and M. Staudt. 1999a. *ConceptBase V5.1 User Manual*. Aachen, Germany: University of Aachen.

Jarke, M., M. Jeusfeld, and M. Staudt. 1999b. *ConceptBase V5.1 Programmer's Manual*. Aachen, Germany: University of Aachen.

Kegel, D. 1996. Dan Kegel's ISDN page. Available at ⟨http://www.alumni.caltech.edu/~dank/isdn/⟩.

Kessler, G. C. 1998. *ISDN: Concepts, Facilities, and Services*. New York: McGraw-Hill.

Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis. 1990. "Telos: A Language for Representing Knowledge about Information Systems." *ACM Transactions on Information Systems* 8, no. 4: 325–362.

Neumann, P. G. 1995. *Computer-Related Risks*. Reading, MA: Addison-Wesley.

Pohl, K. 1994. "The Three Dimensions of Requirements Engineering—A Framework and Its Applications." *Information Systems* 19, no. 3: 243–258.

PTT Telecom. 1993. *User-Network Aspects of Euro-ISDN*. The Hague, Netherlands: PTT Telecom.

Reed, R., W. Bouma, M. M. Marques, and J. Evans. 1992. "Methods for Service Software Design." In *Proceedings of the Eighth International Conference on Software Engineering for Telecommunication Systems and Services*, 127–134. London: IEE Press.

Reed, R., J. De Man, and B. Møller-Pedersen. 1989. "A Formal Techniques Environment for Telecommunications Software." In *Proceedings of the Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, 6–11. London: IEE Press.

Ryan, R. 1994. "The Role of AI in Requirements Engineering." Position paper for the Dagstuhl Workshop on System Requirements: Analysis, Management and Exploitation. Dagstuhl, Germany, October 4–7, 1994.

Schach, S. R. 1999. *Software Engineering*. Homewood, IL: Asken Associates.

SCORE (Service Creation in an Object-Oriented Reuse Environment)—*Methods and Tools, Report on Methods and Tools for Service Creation*, vol. 1, *Service Interaction*. 1995. Deliverable D206-Vol.1, R2017/SCO/WP2/DS/P/031/b1 Race project 2017.

Sommerville, I. 2000. *Software Engineering*. Reading, MA: Addison-Wesley.

Standish Group. 1994. *CHAOS.* Software development report. Available at ⟨http://www.standishgroup.com/sample_research/chaos_1994_1.php⟩.

Thörner, J. 1994. *Intelligent Networks*. Norwood, MA: Artech House.

Wieringa, R., E. Dubois, and S. Huyts. 1997. "Integrating Semi-formal and Formal Requirements." In *Proceedings of the Ninth International Conference on Advance Information Systems Engineering (CAiSE'97)* ed. A. Olive and J. Pastor, 19–32. New York: Springer.

5 Metadata for Hypermedia Textbooks From RDF to O-Telos and Back

Martin Wolpers and Wolfgang Nejdl

5.1 Introduction

The World Wide Web can be viewed as a huge directed graph, with Web pages representing the nodes in the graph and hyperlinks representing directed edges between these nodes. Both nodes and edges are untyped, and additional metadata are needed to lend meaning to the nodes and edges. The metadata must be accessible to machines in order to enable automatic processing. Formalizing and building this metadata is the goal of the Semantic Web initiative, building on standardized metadata for the World Wide Web.

This chapter discusses some modeling aspects of the recommended metadata language standard for the Semantic Web, called the Resource Description Framework (RDF) (Manola and Miller 2004). We will use the O-Telos language (Jeusfeld 1992), which, as noted in chapter 3, is based on Telos (Mylopoulos et al. 1990), to analyze and discuss RDF. First, we explicitly model RDF in O-Telos to point out some RDF characteristics and peculiarities, and second, we translate RDF and its schema, RDF Schema specification (RDFS) (Brickley and Guha 2004), into O-Telos, showing how O-Telos can be used as a superset of RDF(S), providing all the characteristics of RDF plus some additional ones. We also discuss RDF(S) and O-Telos as the basic modeling language used in the KBS Hyperbook system and show how RDF(S) data can be imported into and exported from ConceptBase. Furthermore, the chapter introduces the KBS Hyperbook system, which uses the RDF Schema specification to process and display lecture material on the Web.

5.2 RDF in a Nutshell

RDF is a recommended standard for annotating resources with semantic information that uses underlying conceptual models (schemas) to define the classes and properties employed for these semantic annotations. Beckett (2004) defines the syntax of RDF annotations and Brickley and Guha (2004) define the semantics of RDF in

schemas. Note that in the specification of RDF, some rather unconventional design decisions are made: for example, to give dual roles to the inheritance, instantiation, range, and domain constructs, which are used both as primitive constructs and as specific instances of RDF properties themselves.

RDF schemas define the structure of the metadata describing a particular data model, which consists basically of resources. A resource may be a part of a Web page, an entire Web page, a whole collection of Web pages, an entire Web site, or an object that is not directly accessible via the Web, for example, a printed book. RDF resources are described by a uniform resource identificator (URI), so that each item referenced by a URI is of type `rdfs:Resource`. Note that `rdfs:` specifies the namespace as defined by the XML namespace facility (Bray et al. 2006). The specification for these schemas, the RDF Schema specification, constitutes some basic classes and properties and is claimed to be extendable to fit potentially any given domain.

Lassila and Swick (1999) define RDF and its goals as follows:

> The broad goal of RDF is to define a mechanism for describing resources that makes no assumptions about a particular application domain, nor defines (a priori) the semantics of any application domain. The definition of the mechanism should be domain-neutral, yet the mechanism should be suitable for describing information about any domain.

All resources are grouped in RDF into a well-defined set of classes that are arranged hierarchically. The relationships connecting the classes are expressed by properties. The properties themselves are instances of the class `rdf:Property`.

5.3 Explicitly Modeling RDF in O-Telos

This chapter describes how RDF can be explicitly modeled using O-Telos. RDF is introduced in detail, and some of its peculiarities and special characteristics are pointed out.

The basic RDFS constructs are classes and properties. Classes are arranged hierarchically; the root of this class hierarchy is `rdfs:Resource`, which has `rdfs:Class` as a subclass. The `rdfs:` prefix on the names of these classes indicates that they are part of the RDF Schema specification, whereas the prefix while `rdf:` indicates membership in the RDF data model. The RDF data model defines basic RDF constructs like `rdf:Property` and `rdf:Statement`, whereas semantically motivated RDF constructs are defined in the RDF Schema specification.

Properties connect classes and thus represent relations between these classes. They can be constrained to members of certain classes. Properties are defined by the `rdf:Property` construct and resemble attributes in more conventional metamodels. As such, they are used to describe resources by letting the resources hold values. They are also resources themselves and can be referenced by a unique URI.

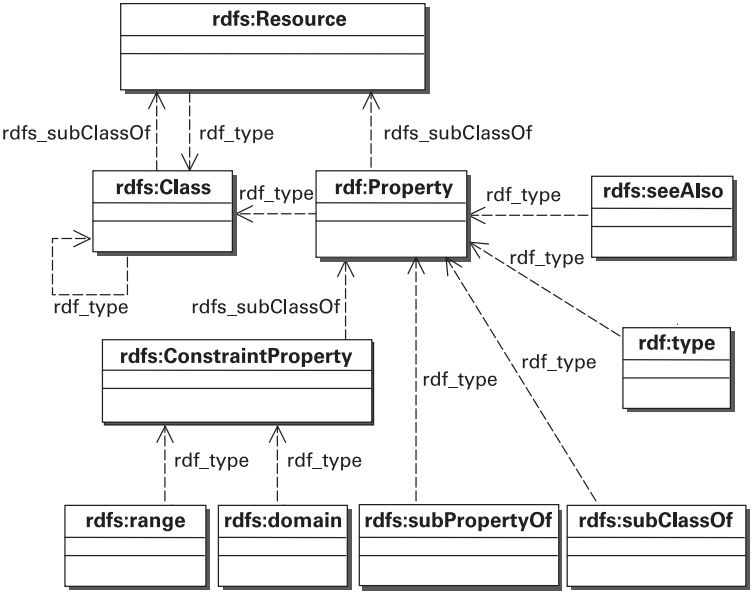


Figure 5.1
Part of the class hierarchy for RDF and RDF Schema

Figure 5.1 shows part of the class hierarchy into which the RDF and RDF Schema constructs are structured. (The figure represents only the constructs necessary for this discussion and their relationships.) `rdfs:Resource` states that the construct instantiating `rdfs:Resource` is found on the Internet. `rdfs:Class` as well as `rdf:Property` are subclasses of `rdfs:Resource`. Furthermore, `rdfs:Resource` as well as `rdf:Property` and its subclass `rdfs:ConstraintProperty` are instances of `rdfs:Class`. Also some of the properties predefined in RDF are included as instances of `rdf:Property`. Thus, the dual role of the properties `rdfs:subClassOf` and `rdf:type` becomes evident. They are instances of `rdf:Property` and at the same time are used as primitive constructs in the hierarchy shown in the figure.

Figure 5.2 shows some of the constraints that are defined in RDF Schema. (A more detailed overview is provided in Brickley and Guha 2004.) The figure shows how the `rdfs:range` and `rdfs:domain` properties restrict members and values of the properties `rdf:type` and `rdfs:subClassOf`. As with the properties `rdf:type` and `rdfs:subClassOf`, the dual role of `rdfs:range` and `rdfs:domain` can be observed in this figure, where they are used both as primitive constraints and at the same time as constraints on themselves.

It is interesting and instructive to compare RDF as a modeling language with the modeling languages O-Telos (Jeusfeld 1992) and Telos (Staudt, Jarke, and Jeusfeld

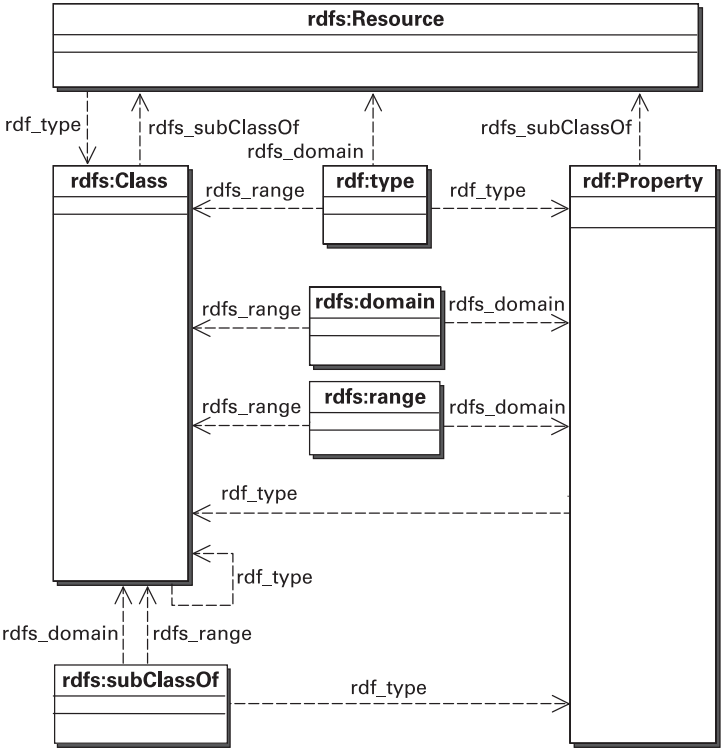


Figure 5.2
Some of the constraints of RDF Schema

1996; Mylopoulos et al. 1990), which have been shown to be very suitable for various modeling and metamodeling tasks. O-Telos specifies as main constructs `Proposition` which are subsumed by `Individual` and `Attribute` (see chapter 3 for a thorough discussion of O-Telos). Because the `Attribute` construct usually relates two or more individuals, the name "relation" seems more suitable for this discussion.

Comparing O-Telos to RDF, one finds that both modeling languages use a rather property-centric (or relation- or attribute-centric) approach for modeling domains. Metadata about domains are stated by declaring properties of resources or individuals. In fact, this is one of the architectural principles of the Web (Berners-Lee 1998).

O-Telos uses two predefined relations, `isA` and `in`. The `isA` relation denotes a class-subclass relationship, with the accompanying inheritance constraints. The `in` relation denotes that a construct is an instance of a class, also accompanied by the appropriate constraints. The `in` relation is used to define a class by stating that an individual is instance of another individual, which then must be defined as a class.

With the aim of expressing the metamodel of the RDF Schema specification in O-Telos, we represent all functional elements of RDF and RDFS in the O-Telos model. In other words, each of the various constructs of RDF has a counterpart in the metamodel formulated in O-Telos.

We simplify the discussion by focusing on the main RDF constructs, as shown in figure 5.1. Furthermore, the dual role of some of the RDF constructs as primitive constructs and as properties is resolved in our O-Telos model. Specifically, we distinguish the different uses of the RDF constructs `rdfs:range`, `rdfs:domain`, `rdfs:subClassOf`, and `rdf:type`. Consequently, our model allows us to distinguish between how RDF annotations are structured and used and the fact that RDF schemas are themselves data described by specific Web resources.

The RDF Schema specification contains predefined properties that are used for simply describing a specific resource, for example, `rdfs:comment` and `rdfs:seeAlso`. These properties are convenient but are not needed for the definition of other properties. RDF Schema also includes the properties `rdfs:subClassOf`, `rdf:type`, `rdfs:range`, and `rdfs:domain`, which have, unlike `rdfs:comment` and `rdfs:seeAlso` properties, certain constraints while being instances of the class `rdf:Property` themselves:

- `rdfs:subClassOf` is a primitive construct describing the functionality of inheritance.
- `rdf:type` is a primitive construct describing the functionality of instantiation.
- `rdfs:range` is a primitive construct describing the range of values a property can hold.
- `rdfs:domain` is a primitive construct defining the classes whose instances can use the property in question.

The dual role of these four properties, being both primitive constructs and instances of primitive constructs, makes it difficult to read and understand the specification.

O-Telos and RDF use the `in` relation (O-Telos), also known as the type relation (RDF), both as a primitive construct and as a regular property (attribute/relation). Both modeling languages use this relation to define that an individual is a class if another individual is an instance of the first individual.

For example, the expression `X in C end` defines `x` as an instance of `c`, which implies that `c` is a class. In order to denote the instantiation, one would use the expression `X->C`. `X->C` itself is an instance of `Proposition!InstanceOf`. Note that the frame `in in property end` is not possible, because the *label* of the instance relation `in` is not an object itself. In the same way, O-Telos imposes constraints on the other RDF constructs that hold dual roles, like `rdfs:subClassOf`, `rdf:range`, and `rdf:domain`, so their dual roles cannot be modeled straightforwardly in O-Telos.

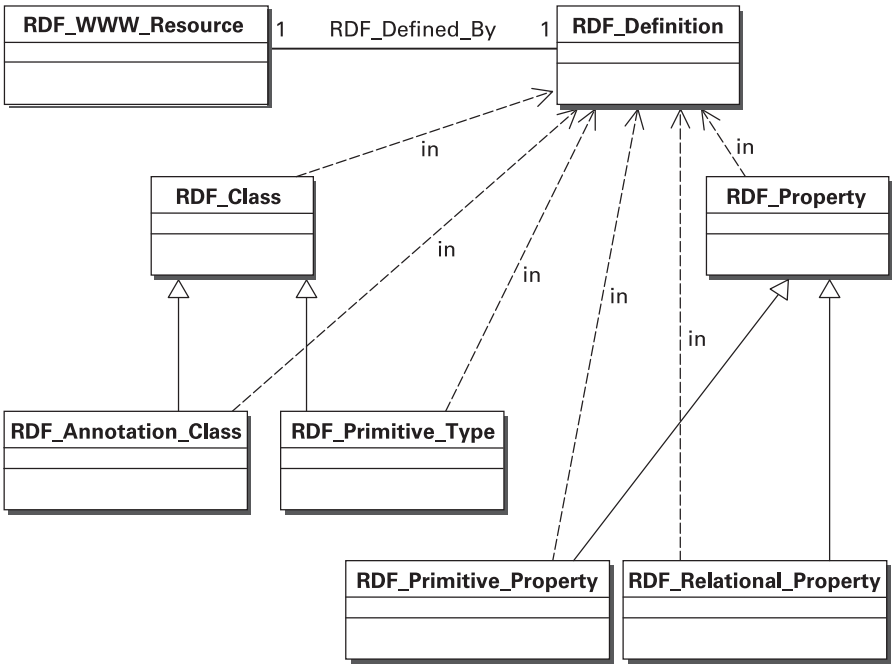


Figure 5.3
Basic RDF constructs as instances of `RDF_Definition`

The dual role of the aforementioned properties is resolved through the explicit modeling of their meaning separately from their function. Therefore we represent the `rdfs:subClassOf` property by the `RDF_Subclass` class, which is not an instance of the `RDF_Property` class. We describe this separation in more detail later in this section. Furthermore, we introduce `RDF_Definition` and `RDF_WWW_Resource` and their relation `RDF_Defined_By`, as shown in figure 5.3. These constructs represent the RDF philosophy that every definition (represented by `RDF_Definition/rdfs:Resource`) is itself described by a resource. Figure 5.4 represents the part of the schema specification that defines the main RDF constructs used for the construction of RDF schemas and how these constructs are used for annotating Web resources, thus resolving the dual role of the aforementioned properties.

The familiar constructs `RDF_Class` and `RDF_Property` as shown in figure 5.4 correspond to the `rdfs:Class` and `rdf:Property` constructs, respectively. They are related through the relations `RDF_Domain` and `RDF_Range`, which correspond to `rdfs:domain` and `rdfs:range`, respectively. However, both relations `RDF_Domain` and `RDF_Range` are specifically defined by this specification and are therefore not

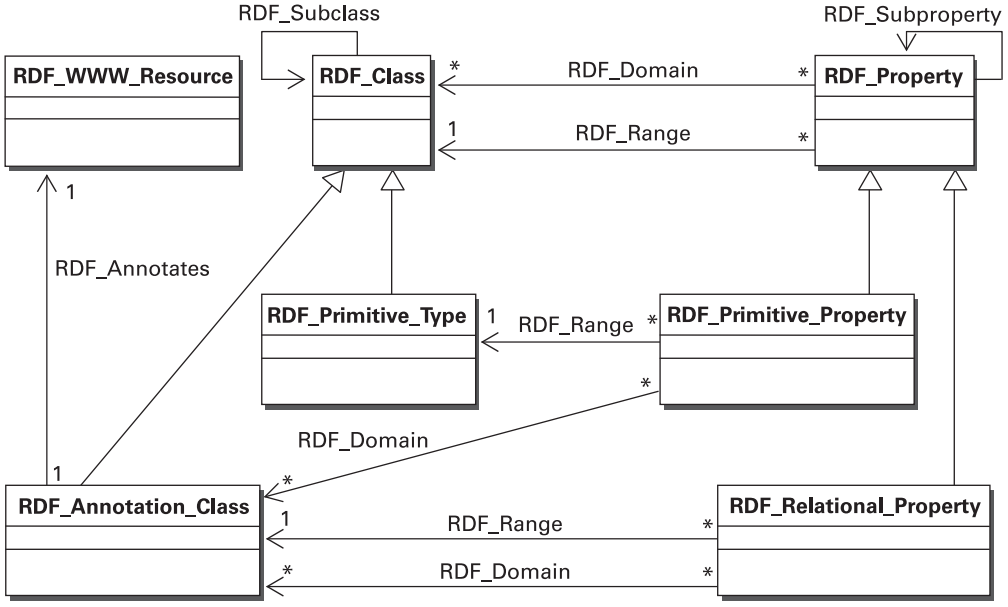


Figure 5.4
Main constructs of the RDF Schema specification formulated in O-Telos

instances of `RDF_Property` (in contrast to what is the case in the RDF Schema specification).

Similarly the relations `RDF_Subclass` and `RDF_Subproperty` substitute for the RDF constructs `rdfs:subClassOf` and `rdfs:subPropertyOf`, respectively. They have the same semantics as the metalanguage construct `isA` in O-Telos, so they enable class-subclass and property-subproperty relationships, respectively, to be defined. Because of their explicit definition they are not instances of RDF-Property.

Because it carries two meanings, the `rdf:type` construct needs some special attention when it is expressed in O-Telos. For cases in which it expresses the relationship between RDF metadata and a conventional Web resource, the O-Telos relationships `RDF_Annotates` and `RDF_Defined_By` substitute for `rdf:type`. For cases in which `rdf:type` represents a conventional instantiation, the O-Telos `in` construct, denoting instantiation, substitutes for it.

`RDF_Class` has two subclasses, `RDF_Annotation_Class` and `RDF_Primitive_Type`. `RDF_Annotation_Class` (with its further subclasses like `RDF_SeeAlso`, the O-Telos counterpart to the `rdf:seeAlso` property) has objects as instances that, because they correspond to RDF statements, are used to annotate Web resources (instances of `RDF_WWW_Resource`). Of course, an instance of `RDF_WWW_Resource`

can be annotated by more than one instance of `RDF_Annotation_Class` (i.e., when different RDF schemas are used to define the metadata for a specific Web resource).

Inheritance and instantiation functionality is expressed by using `isA` and `in` O-Telos statements, thus resolving the dual role of `rdf:type` and `rdfs:subClassOf`. Also, the primitive type `rdfs:Literal`, which takes on String values, is replaced by the primitive O-Telos class `String`. `RDF_Primitive_Type`, with its subclasses like `String`, has as instances specific values for `RDF_Property`, of type `String`, `Integer`, etc. (often predefined types and properties). It is clear that a property relating several objects like the `rdfs:seeAlso` construct is represented as a subclass of the `RDF_Relational_Property`. The same is true for properties that hold values, like `rdfs:label`. These are primitive properties because they assign, for example, a `String` (a primitive type) to an object, thus describing the definition of an attribute, which has a name (here, `rdfs:seeAlso`) and a value (here a `String`).

5.4 Directly Mapping RDF(S) to O-Telos

In contrast to the modeling approach described in the foregoing, in which all RDF constructs are modeled explicitly, this chapter directly maps RDF(S) to O-Telos; we call the result O-Telos-RDF. The mapping relies heavily on the definition of O-Telos, explicitly modifying the O-Telos axioms to suit the intentions of RDF(S). The mapping also facilitates the translation of RDF into O-Telos, thus enabling RDF metadata to be imported into a ConceptBase database.

The example used throughout this section is a small database containing the RDF description of some books, and displays the metadata for four books that we use in our lectures. To simplify our presentation, we use only the properties `rdf:type`, `dc:title`, and `dc:author` (`rdf:` denoting the RDF namespace and `dc:` denoting the Dublin Core namespace [Weibel et al. 1998]). But of course our translation encompasses all RDF(S) properties, including `rdf:subClassOf` and `rdf:subPropertyOf`.

As noted previously, the translation of RDF metadata into O-Telos also means that we can import RDF metadata into ConceptBase. Our translation is based on the discussion of O-Telos-RDF in Nejdl, Dhraief, and Wolpers 2001, which analyzes similarities and differences between RDF(S) and O-Telos.

Our translation ensures that RDF(S) data are translated without any loss of information into O-Telos data. We use the representation of RDF expressions as triple statements as our starting point. Each triple is extended to a quadruple with a unique ID (as O-Telos extends the RDF triples by a fourth argument, the triple ID). Additional quadruples not originally present in RDF(S) are included as well, as O-Telos-RDF employs explicit instantiation of properties, not only of resources/

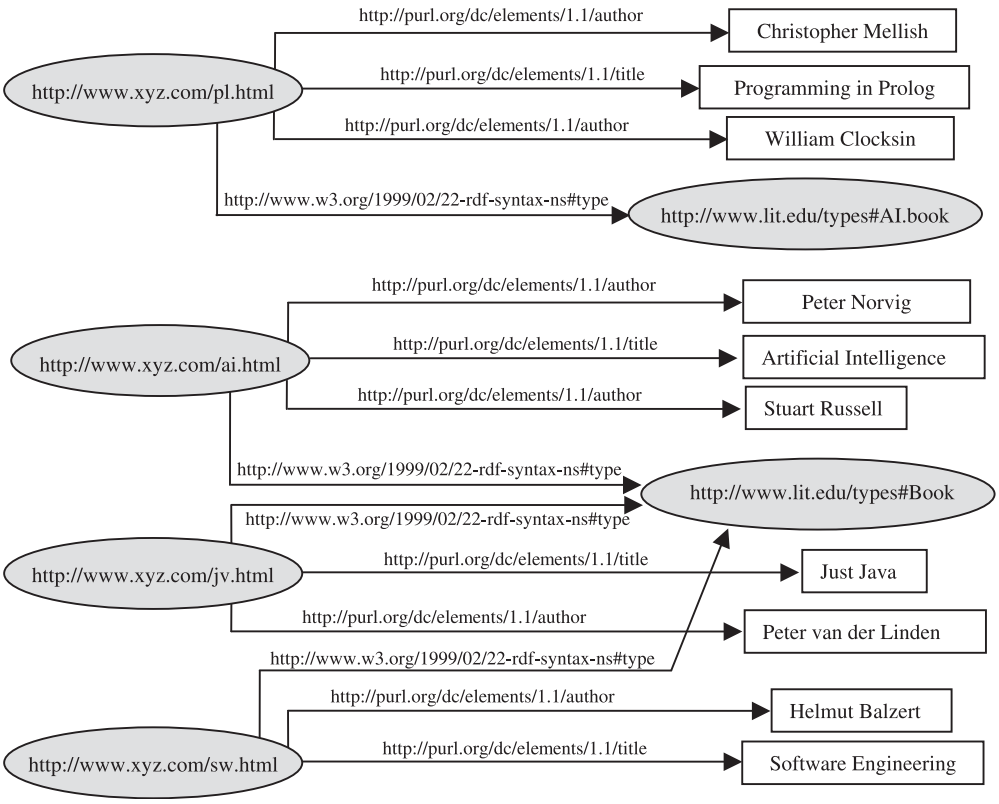


Figure 5.5
RDF graph of the example book database

objects (Nejdl, Dhraief, and Wolpers 2001). This set of quadruples then forms the O-Telos database, with each quadruple representing an O-Telos statement.

To give an example of the translation process, we focus on just one RDF resource description. The translation of the other resource descriptions of our example book database (represented graphically in figure 5.5) follows basically the same procedure. (See chapter 3 for a thorough discussion of O-Telos and its various representations.) The RDF statements for the example resource `http://www.xyz.com/jv.html` describe a book with its properties `Just Java` of type `dc:title` and `Peter van der Linden` of type `dc:author`. The resource itself is of the `rdf:type http://www.lit.edu/types#Book`. The RDF statements are as follows:

```
(http://www.xyz.com/jv.html,
http://www.w3.org/1999/02/22-rdf-syntax-ns#type,
http://www.lit.edu/types#Book)
```

```
(http://www.xyz.com/jv.html,
http://purl.org/dc/elements/1.1/title,
"Just Java")

(http://www.xyz.com/jv.html,
http://purl.org/dc/elements/1.1/author,
"Peter van der Linden")
```

In the translation of these RDF triples into O-Telos quadruples that follows, we omit the correct representation of strings for simplicity reasons. In O-Telos strings are objects themselves and are represented as such. Also, we have additional tuples for all resources/objects and instantiation statements for them, as well as additional instantiation statements for properties. Furthermore, in our example the IDs are represented by a short form, `sidx`, to enhance readability. These abbreviations expand to something of the form `namespace:resourcename`. Usually the IDs are the URIs of the namespace plus the respective RDF construct; for example, `sid1` is the abbreviated form for the resource `http://www.xyz.com/jv.html` of the RDF triples in the foregoing. The RDF triples written as O-Telos quadruples are as follows:

```
P(sid1, sid1, jv_html, sid1)
P(sid2, sid1, In, #Book)
P(sid3, sid1, In, #DCElements11)
P(sid4, sid1, In, #Individual)
P(sid5, sid1, title1, "Just Java")
P(sid6, sid5, In, #DCElements11.title)
P(sid7, sid5, In, #Attribute)
P(sid8, sid1, author1, "Peter van der Linden")
P(sid9, sid8, In, #DCElements11.author)
P(sid10, sid8, In, #Attribute)
P(sid11, sid1, namespace, http://www.xyz.com/)
```

We now return to the description of the translation. The focus is on the axioms describing statement IDs, instantiations, and properties, which we explain further in the following (our explanation is taken from Nejdl, Dhraief, and Wolpers 2001 and is based on Jeusfeld 1992). Most other aspects of RDF(S) are defined basically in the same way as in O-Telos, so we do not repeat the descriptions here.

5.4.1 Statement IDs and Individuals

Each tuple in O-Telos includes a unique identifier as its statement ID, in contrast to RDF triples, which can be referenced only using the (somewhat clumsy) reification mechanism. This identifier is invisible in O-Telos to the user, though the user can uniquely reference each statement by means of the appropriate combination of sub-

ject, predicate, and object of the tuple (depending on what kind of statement he wants to reference).

In order to keep the discussion simple, we only state here those axioms that are either modifications of O-Telos axioms or of particular importance in the discussion. (A complete list of O-Telos axioms can be found in section 3.8.)

Axiom 5.1 (Corresponds to axiom 3.1) Statement identifiers uniquely identify statements.

```
forall sid,x1,x2,l1,l2,y1,y2
P(sid,x1,l1,y1) and P(sid,x2,l2,y2) ==> (x1=x2) and (y1=y2) and
(l1=l2)
```

Axiom 5.2 (Corresponds to axiom 3.2) If the label of an individual is an atom, it is unique within its namespace. Together with its namespace, or if the label is already a URI, it is globally unique. Therefore the statement ID of an individual is globally unique in all cases.

The generation of the identifiers of our O-Telos tuples must be based on O-Telos-RDF axioms 5.1 and 5.2, which state that each RDF statement must obtain a unique ID to form the O-Telos statement. We use these IDs as statement identifiers, which are globally unique.

5.4.2 Properties

Attributes in O-Telos and O-Telos-RDF have the same structure as RDF properties but are instantiated explicitly from their definition using an explicit type statement and a new name for the instantiated property (in contrast to RDF, in which the property name is used directly as a predicate in the instantiated property).

Axiom 5.3 (Definition of properties; corresponds to axiom 3.4)

```
forall sid,x,p,y
P(sid,x,p,y) and (sid \= x) and (sid \= y) and (p \= subClassOf) and
(p \= subPropertyOf) and (p \= type)
<==> type(sid,otelos:property)
```

Axiom 5.4 (Corresponds to axiom 3.3) Names of "object-scoped" properties are unique in conjunction with the source object.

```
forall sid1,sid2,x,p,y1,y2
P(sid1,x,p,y1) and P(sid2,x,p,y2)
==> (sid1=sid2) or (p=type) or (p=subClassOf) or (p=subPropertyOf)
```

Axiom 5.5 (Corresponds to axiom 3.3) For "globally scoped" properties, axiom 5.4 is extended, so that the names of attributes are unique even without conjunction with the source.

```
forall sid1,x1,x2,p,y1,y2
P(ns:p,x1,p,y1) and P(sid1,x2,p,y2)
==> ((sid1=ns:p) and (x1=x2) and (y1=y2)) or (p=type) or
(p=subClassOf) or (p=subPropertyOf)
```

RDF properties are translated into O-Telos attributes according to axioms 5.3, 5.4, and 5.5. (We use the terms "property" and "attribute" synonymously in this context.) According to axiom 5.3, all O-Telos attributes are instances of the O-Telos object `Attribute`. (Note that `Attribute` and `otelos:property` denote the same construct.) In the translation process, all these statements have to be generated, as they are not part of the original RDF representation. For example, in the examples earlier in the section, the statements `sid5` to `sid7` define the second RDF triple.

Axioms 5.4 and 5.5 state that each attribute is declared in the context of a specific subject. If no subject exists for a particular attribute, the attribute is defined as an instance of the class `Attribute`. As an example, statements `sid6` and `sid7`, declare the `title` attribute as an instance of `#DCElements11.title` and `#Attribute`, respectively. The `rdfs:domain` and `rdfs:range` constructs used to declare RDF properties are implicitly included in O-Telos attribute declarations.

Axiom 5.6 (Corresponds to axiom 3.17) In case x is an instance of two classes, c and d, both of which define a property m, x also has to be an instance of a class g, which is a subclass of both c and d and which also defines property m:

```
forall x,m,y,c,d,sid1,sid2,e,f
(type(x,c) and type(x,d) and P(sid1,c,m,e) and P(sid2,d,m,f)
==> exists g,sid3,h type(x,g) and P(sid3,g,m,h) and subClassOf(g,c)
and subClassOf(g,d))
```

Further axioms determine the translation for `rdf:type`. Each `rdf:type` statement is expressed in O-Telos as an instantiation statement. A special case is the instantiation from more than one class that declares a property with the same name. Here the translation process has to build an explicit property declaration of each property according to axiom 5.6.

Axioms 5.1 to 5.6 illustrate briefly some of the fundamentals of the translation process. In addition to these representational issues, we have to deal with more practical problems, such as different namespaces or strictly enforced ConceptBase character and naming limitations.

An example of the O-Telos frame representation is the frame of the book `Just Java`:

```
Individual jv_html in Book, DCElements with
  title
    title1 : "Just Java"
```

```
  author
    author1 : "Peter van der Linden"
  attribute
    namespace : "http://www.xyz.com/"
end
```

If an RDF statement includes elements from other than the current schema, these elements are grouped in specially created O-Telos classes. For example the properties `dc:title` and `dc:author` of the `Just Java` book's RDF description, originating in the Dublin Core schema, are grouped in an O-Telos class called `DCElements11` (see statements `sid5` and `sid6` from the example earlier in this section, which define the attribute `title` as an instance of the attribute `title` of class `DCElements11`).

The O-Telos attribute `namespace` is introduced to hold the namespace of each RDF resource and property. Usually the namespace is part of the unique ID of each statement, as can be seen in the O-Telos quadruple example earlier in the section. Unfortunately, the O-Telos frame syntax and parser prohibit any special characters of uniform resource locators (URLs) in the element name. Therefore, we introduce the `namespace` attribute, which is assigned to each element and attribute. This attribute stores the namespace or resource URL as a workaround. As an example, compare the frame representation and the quadruples of the book `Just Java`.

Figure 5.6 shows the O-Telos graph of the same part of the book database that is shown as an RDF graph in figure 5.5. Comparison of the two graphs shows their strong similarities. The chief difference is in the notations and serializations (triple vs. quadruple and XML vs. frame syntax) that they employ. Using the approach described in this section we can easily express RDF(S) data in the O-Telos language and import them into ConceptBase.

5.5 An Application of O-Telos

The KBS Hyperbook System (Fröhlich, Henze, and Nejdl 1997; Fröhlich, Nejdl, and Wolpers 1998; Henze et al. 1999; Henze and Nejdl 1999; Nejdl and Wolpers 1998, 1999) is an open, interactive hypermedia system designed to fulfill the needs of lecturers and students. Conceptual modeling being only one of its focuses, the system also implements ideas regarding user adaptation, teaching strategies, and so on. We do not discuss these issues here (see the cited references instead) but focus on the use of conceptual modeling.

The KBS Hyperbook System is used to build hyperbooks. A *hyperbook* is defined as

"an information repository, which integrates a set of (possibly distributed) information sources using explicit semantic models and metadata" (Henze et al. 1999, 26).

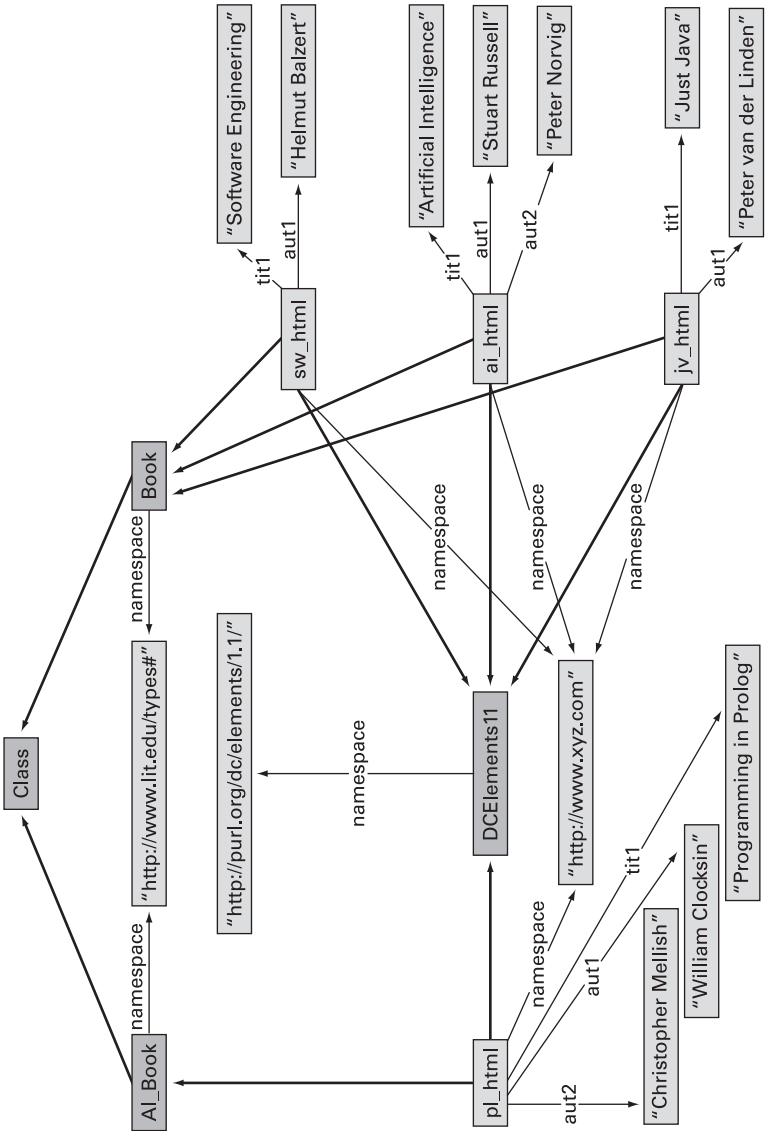


Figure 5.6
O-Telos graph of the example book database

The information sources in this definition contain resources. A resource, as mentioned in section 5.2, may be a Web page, part of a Web page, a collection of Web pages, a Web site, or something that is not Web-accessible, for example, a Word document on an intranet. KBS Hyperbook uses semantic models and metadata to structure and display resources from such information sources. It employs a general representation model that defines concepts relevant to hyperbooks in general. This representation model serves as a basic language from which all further conceptual models for a wide variety of purposes are built.

These conceptual models (actually metadata models) are then used to annotate, structure, and integrate external data. Therefore, in the case of external data on the World Wide Web—Web pages—the conceptual models take on the role of information indices that determine the navigational structures of these pages. Thus, the conceptual models enable various views of the described Web pages. The conceptual models also serve as a schema for maintenance of information, thus providing rules for the integration and deletion of information (comparable to the role of a database schema). Furthermore, they allow the specification of arbitrary metadata and structural information.

Additional presentation classes govern resource layout. Semantic relationships among resources can be modeled according to the system's general representation model and displayed as indices, links, or sequences, together with the corresponding resource. Thus, the KBS Hyberbook System can be used also as a tool for viewing and browsing semantic models.

The distinction in KBS Hyperbook between the general representation model and the conceptual models is important. The fact that general hyperbook concepts are modeled in the general representation model means that the conceptual models need to deal only with concepts relevant to their application domains and do not need to deal with hyperbook concepts and functionalities. The models avoid any self-reference by clearly stating primitive constructs only in the metadata model and by basing all conceptual models on these primitive constructs. This simplifies the formalization of the domain and makes the modeling approach similar to conventional metamodeling approaches such as IRDS (ISO/IEC 1990).

The RDF Schema specification formulated in O-Telos and described in the previous section defines a representation model that meets the requirements of the KBS Hyperbook System. It is therefore possible to use this definition as a representation model and display the semantic relationships as well as the related resources.

Figure 5.7 shows how the KBS Hyperbook System visualizes a conceptual model that describes resources (in this case, Web pages) that are used for an introductory computer science course at the Universities of Hannover in Germany and Bozen in Italy. A specific resource, here the Web page named `Bingo! Erste Analyse`, is shown in the right frame of a browser window. This resource has several

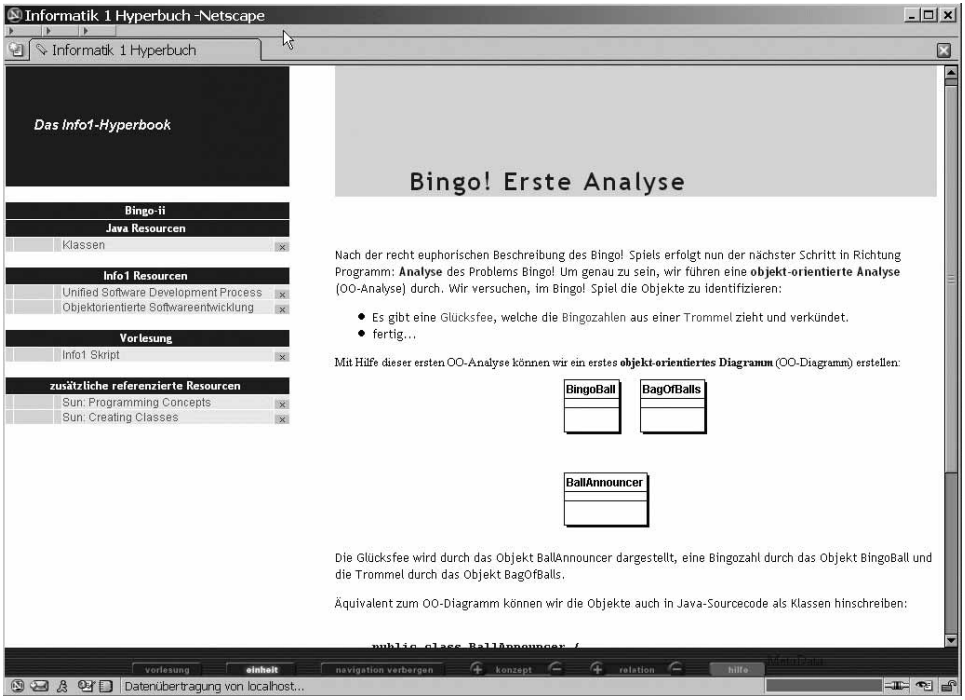


Figure 5.7
KBS Hyperbook, using the RDF Schema specification

relations to other resources, according to the conceptual model. These bidirectional relations are shown in the left frame as grouped unidirectional links; that is, this resource has relations to other `Info 1 Resources` as well as to `Java Resources`, and so on.

As we see in figure 5.7, the KBS Hyperbook System is able to use a conceptual model to build a navigational structure for each resource that might be explored by such tools as regular Web browsers. Figure 5.8 shows another feature of the KBS Hyperbook System. In that figure, all relations of a conceptual model are displayed as grouped links in the left frame. The right frame displays the Web pages that these links point to. In this scenario, KBS Hyperbook works as an information index to a set of resources; the resources are Web pages and the index entries (relations) are displayed as links.

Both figures show a third frame at the bottom of the browser window. This frame contains some controls that enable the conceptual model to be manipulated. The controls enable concepts and relations to be added and removed at the conceptual and data levels. They basically trigger the insertion and removal of O-Telos frames

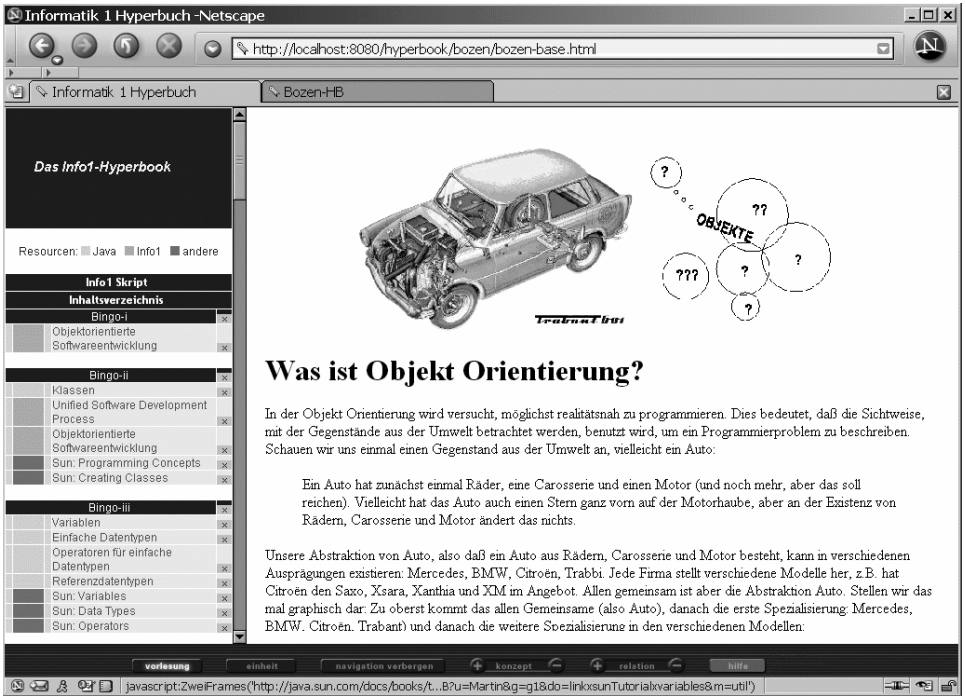


Figure 5.8
The KBS Hyperbook user interface

into the system. Since the controls aren't the subject of this chapter, we omit their discussion here.

5.5.1 Modeling the Lecture "Artificial Intelligence"

In this section we model the lecture "Artificial Intelligence" by one of the authors at the University of Hannover. Based on the metamodel and the O-Telos-RDF translation, we show how the model expressed in RDF(S) is translated into O-Telos, thus enabling the KBS Hyperbook System to generate and display navigational structures of the model and Web resources. A part of the RDF model for this lecture is given here:

```
<rdf:RDF xml:lang="en"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:s="http://www.kbs.uni-hannover.de/otelos/2001/06/example-
  schema#">
```

```
<rdf:Description ID="Lecture">
  <rdf:type resource="rdfs:Class"/>
</rdf:Description>

<rdf:Description ID="lectureUnits">
  <rdf:type resource="rdf:Property"/>
  <rdfs:range resource="s:LectureUnit"/>
  <rdfs:domain resource="s:Lecture"/>
</rdf:Description>

<rdf:Description ID="LectureUnit">
  <rdf:type resource="rdfs:Class"/>
</rdf:Description>

<rdf:Description ID="title">
  <rdf:type resource="rdf:Property"/>
  <rdfs:range resource="rdfs:Literal"/>
  <rdfs:domain resource="s:LectureUnit"/>
</rdf:Description>

<rdf:Description ID="description">
  <rdf:type resource="rdf:Property"/>
  <rdfs:range resource="rdfs:Literal"/>
  <rdfs:domain resource="s:LectureUnit"/>
</rdf:Description>

<rdf:Description ID="parentCourse">
  <rdf:type resource="rdf:Property"/>
  <rdfs:range resource="s:Lecture"/>
  <rdfs:domain resource="s:LectureUnit"/>
</rdf:Description>

<rdf:Description ID="theoryPage">
  <rdf:type resource="rdf:Property"/>
  <rdfs:range resource="s:TheoryUnit"/>
  <rdfs:domain resource="s:LectureUnit"/>
</rdf:Description>

<rdf:Description ID="parentUnit">
  <rdf:type resource="rdf:Property"/>
  <rdfs:range resource="s:LectureUnit"/>
  <rdfs:domain resource="s:TheoryUnit"/>
</rdf:Description>
```

```
  <rdf:Description ID="TheoryUnit">
    <rdf:type resource="rdfs:Class"/>
  </rdf:Description>

  <rdf:Description ID="AILecture">
    <rdf:type resource="s:Lecture"/>
    <lectureUnits resource="s:LectureUnit1"/>
  </rdf:Description>

  <rdf:Description ID="LectureUnit1">
    <rdf:type resource="s:LectureUnit"/>
    <title> Lecture Unit 1</title>
    <description>
      Introduction to intelligent agents
    </description>
    <parentCourse resource="s:AILecture"/>
    <theoryPage resource="http://.../Definitions.htm"/>
    <theoryPage resource="http://.../Characterisation.htm"/>
    <theoryPage resource="http://.../Structure.htm"/>
    <theoryPage resource="http://.../Types.htm"/>
  </rdf:Description>

  <rdf:Description about="http://.../Definitions.htm">
    <rdf:type resource="s:TheoryUnit"/>
    <parentUnit resource="s:LectureUnit1"/>
  </rdf:Description>

  <rdf:Description about="http://.../Characterisation.htm">
    <rdf:type resource="s:TheoryUnit"/>
    <parentUnit resource="s:LectureUnit1"/>
  </rdf:Description>

  <rdf:Description about="http://.../Structure.htm">
    <rdf:type resource="s:TheoryUnit"/>
    <parentUnit resource="s:LectureUnit1"/>
  </rdf:Description>

  <rdf:Description about="http://.../Types.htm">
    <rdf:type resource="s:TheoryUnit"/>
    <parentUnit resource="s:LectureUnit1"/>
  </rdf:Description>
</rdf:RDF>
```

To start with, we present the RDF model of the lecture "Artificial Intelligence." The model is quite large, so we limit the example to just the small part shown in the foregoing. The example shows the lecture "Artificial Intelligence," modeled as instance `AILecture` of the class `Lecture`. The lecture consists of various lecture units, stated in the property `lectureUnits`. The example shows only the lecture unit `LectureUnit1` with the `title "LectureUnit1"`. This lecture unit deals with an introduction to intelligent agents (the `description` property). Several theory pages explain the knowledge stated there (the `theoryPage` property).

The modeling of the lecture uses a rather simple metamodel that defines a class `Lecture` that relates to a class `LectureUnit`. Lecture units have various properties, among them a title and a description. They relate to theory pages via the property `theoryPage`. Theory pages (class `TheoryUnit`) describe the thematic theories used in the lecture unit.

Note that the example also declares the complementary relation for the `parentCourse` and the `theoryPage` properties. The class `TheoryUnit` defines the property `parentUnit` relating to the respective lecture unit, whereas the property `parentCourse` of `LectureUnit` relates to the respective lectures, here to `"AILecture"`. At the time the initial design of this metamodel was completed, these complementary relations were necessary, because no inference engine for RDF(S) existed. Since then, however, several such inference engines based on well-defined subsets of RDF for ontological knowledge representation (e.g., TRIPLE [Sintek and Decker 2001]) have emerged, superseding these relations.

In order to include the lecture model within KBS Hyperbook and the underlying ConceptBase database, it must be translated to O-Telos. Using the O-Telos-RDF translation, we derive the O-Telos frames given here, for the same part of the lecture as was shown in the RDF model previously (note that we don't state the `namespace` attributes that belong to the theory pages, for reasons of simplicity):

```
Lecture in Class with
  attribute
    lectureUnits: LectureUnit
end

LectureUnit in Class with
  attribute
    title: String;
    description: String;
    theoryPage: TheoryUnit
  feature
    parentCourse: Lecture
```

```
  rule
    inferParentCourse :
      $ forall lu/LectureUnit, le/Lecture
      (le lectureUnits lu) => (lu parentCourse le) $
end

TheoryUnit in Class with
  feature
    parentUnit: LectureUnit
  rule
    inferParentUnit : $ forall tu/TheoryUnit, lu/LectureUnit
    (lu theoryPage tu) => (tu parentUnit lu) $
end

AILecture in Lecture end

LectureUnit1 in LectureUnit with
  title t: "LectureUnit 1"
  description d: "Introduction to intelligent agents"
  theoryPage
    tp1: Definitions_htm;
    tp2: Characterisation_htm;
    tp3: Structure_htm;
    tp4: Types_htm
end

Definitions_htm in TheoryUnit end
Characterisation_htm in TheoryUnit end
Structure_htm in TheoryUnit end
Types_htm in TheoryUnit end
```

We omit the attributes `parentUnit` and `parentCourse` for the lecture and theory units because they are inferred automatically by the two rules `inferParentUnit` and `inferParentCourse`. The simple logic language Datalog is used to state both rules. The rule `inferParentCourse`, defined in the class `LectureUnit`, states the transitive closure of the attribute `lectureUnits` of class `Lecture`. This means that the attribute `parentCourse` of a given lecture unit relates to the same lectures whose attribute `lectureUnits` relates to the lecture unit. The same holds for the `inferParentUnit` rule, thus providing objects for the respective attribute `parentUnit`.

In this brief section we have given an example of how conceptual models can be used to enable access to information resources. Based on these conceptual models,

KBS Hyperbook can be used to browse and explore the navigational structures of the resources and to view the modeled content. In modeling the "Artificial Intelligence" lecture, we have provided an example of how RDF can be translated for use in a ConceptBase database.

5.6 Summary

The two modeling approaches discussed in the previous sections show how metamodels and conceptual models designed in the context of the World Wide Web can be expressed and formalized using the O-Telos language and implemented employing the ConceptBase database.

The chapter's discussion of the RDF Schema specification points out several unusual characteristics of RDF Schema in comparison to more conventional metadata models. The modeling decision to let the properties `rdfs:subClassOf`, `rdf:type`, `rdfs:domain`, and `rdfs:range` take on dual roles as primitive constructs and as specific instances of RDF properties in the definition makes the specification difficult to understand, and we introduce an alternative model that allows an explicit distinction to be made between modeling and metamodeling features and enables the different meanings of constructs like `rdf:type` to be made explicit. The model is expressed using the O-Telos modeling language.

Through examination of RDF(S) purely as a formal modeling language, another set of interesting insights into RDF(S) is revealed. We present another approach modeling RDF(S) in O-Telos by directly mapping RDF(S) to O-Telos. The resulting O-Telos-RDF dialect has the advantage of allowing easy reification of statements so that they can be used in metamodeling applications, where more than three abstraction hierarchies are needed. Furthermore, the O-Telos approach allows the definition of properties with the same name for different domains that have different ranges (which is not possible in RDF(S)).

As this approach represents a strict extension of RDF(S), all RDF(S) definitions can be translated into O-Telos statements. Translation from O-Telos into RDF(S) is only possible of course, if no O-Telos-specific extensions are used (for example, more than three abstraction levels).

Using explicit metadata about Web pages and other resources, reusing resources becomes much easier. By formalizing and representing the models of the KBS Hyperbook System in RDF, we show that models can be easily exchanged with other RDF-capable systems. Furthermore, semantic relationships between resources are made explicit in the models and can therefore facilitate advanced information needs. For example, models are used as underlying navigational structures within the KBS Hyperbook System.

Acknowledgments

We would like to thank Hadhami Dhraief and Ingo Brunkhorst for many very fruitful hours of discussion. Without their help, this chapter would have not been what it is now.

References

Beckett, D., ed. 2004. "RDF/XML Syntax Specification." Recommendation, W3C. Available at ⟨http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/⟩.

Berners-Lee, T. 1998. "What the Semantic Web Can Represent." Available at ⟨http://www.w3.org/DesignIssues/RDFnot.html⟩.

Bray, T., D. Hollander, A. Layman and R. Tobin, eds. 2006. "Namespaces in XML 1.0 (Second Edition)" Recommendation, W3C. Available at ⟨http://www.w3.org/TR/REC-xml-names/⟩.

Brickley, D., and R. V. Guha, eds. 2004. "RDF Vocabulary Description Language 1.0: RDF Schema." Recommendation, W3C. Available at ⟨http://www.w3c.org/TR/rdf-schema/⟩.

Fröhlich, P., N. Henze, and W. Nejdl. 1997. "Meta Modeling for Hypermedia Design." In *Proceedings of the Second IEEE Metadata Conference*, ed. R. Musick and C. Miller. Available at ⟨http://www.kbs.uni-hannover.de/Arbeiten/Publikationen/1997/metadata/pfroehlich.html⟩.

Fröhlich, P., W. Nejdl, and M. Wolpers. 1998. "KBS Hyperbook—an Open Hyperbook System for Education." In *Proceedings of the Tenth World Conference on Educational Multimedia and Hypermedia (ED-MEDIA'98)*, ed. T. Ottman and I. Tomek. Charlottesville, VA: Association for the Advancement of Computing in Education. (Electronic proceedings)

Henze, N., K. Naceur, W. Nejdl, and M. Wolpers. 1999. "Adaptive Hyperbooks for Constructivist Teaching." *Künstliche Intelligenz* 4: 26–31.

Henze, N., and W. Nejdl. 1999. "Student Modeling for the KBS Hyperbook System Using Bayesian Networks." Technical report, University of Hannover. Available at ⟨http://www.kbs.uni-hannover.de/paper/99/adaptivity.html⟩.

ISO (International Organization for Standardization)/IEC (International Electrotechnical Commission). 1990. "Information Technology—Information Resource Dictionary System (IRDS)—Framework." ISO/IEC International Standard 10027. Geneva: ISO.

Jeusfeld, M. 1992. *Änderungskontrolle in deduktiven Objektbanken*. St. Augustin, Germany: Infix-Verlag.

Lassila, O., and R. Swick, eds. 1999. "Resource Description Framework (RDF) Model and Syntax Specification." W3C. Available at ⟨http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/⟩.

Manola, F., and E. Miller, eds. 2004. "W3C Resource Description Framework (RDF) Model and Syntax Specification." W3C Working Group, Amsterdam. Available at ⟨http://www.w3.org/TR/REC-rdf-syntax/⟩.

Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis. 1990. "Telos: A Language for Representing Knowledge about Information Systems." *ACM Transactions on Information Systems* 8, no. 4: 325–362.

Nejdl, W., H. Dhraief, and M. Wolpers. 2001. "O-Telos-RDF: A Resource Description Format with Enhanced Meta-modeling Functionalities Based on O-Telos." Presented at the Workshop on Knowledge Management and Semantic Annotation at the First International Conference on Knowledge Capture (K-CAP 2001), October 21–23, 2001, Victoria, British Columbia.

Nejdl, W., and M. Wolpers. 1998. "KBS Hyperbook—A Data-Driven Information System on the Web." Technical report, KBS Institute, University of Hannover.

Nejdl, W., and M. Wolpers. 1999. "KBS Hyperbook—A Data-Driven Information System on the Web." Poster Proceedings of the Eighth International World Wide Web Conference, 26–31. Reston, VA: Foretec Seminars.

Sintek, M., and S. Decker. 2001. "TRIPLE—An RDF Query, Inference, and Transformation Language." Workshop on Deductive Databases and Knowledge Management (DDLP'2001) at the International Conference on Applications of Prolog (INAP'2001), Japan, October.

Staudt, M., M. Jarke, and M. Jeusfeld. 1996. "ConceptBase 4.1 User Manual." Technical report, RWTH, Aachen, Germany. Available at ⟨http://www-i5.informatik.rwth-aachen.de/CBdoc/userManual/⟩.

Weibel, S., J. Kunze, C. Lagoze, and M. Wolf. 1998. "Dublin Core Metadata for Resource Discovery." Request for Comments no. 2413, Network Working Group, Internet Engineering Task Force. Available at ⟨http://www.ietf.org/rfc/rfc2413.txt⟩.

6 Monitoring Requirements Development with Goals

William N. Robinson

Managing the development of software requirements can be a complex and difficult task. The environment is often chaotic. As analysts and customers leave the project, they are replaced by others who drive development in new directions. As a result, inconsistencies arise. Newer requirements introduce inconsistencies with older requirements. The introduction of such requirements inconsistencies may violate stated goals of development. In this chapter, techniques are presented that manage requirements document inconsistency by managing inconsistencies that arise between requirements development goals and requirements development enactment.

A specialized development model, called a requirements dialog meta-model, is presented. The metamodel defines a conceptual framework for dialog goal definition, monitoring, and in the case of goal failure, dialog goal reestablishment. The requirements dialog metamodel is supported by an automated multi-user World Wide Web environment, called DealScribe.

This research supports the conclusions that: (1) an automated tool that supports the dialog metamodel can automate the monitoring and reestablishment of formal development goals, (2) development goal monitoring can be used to determine statements of a development dialog that fail to satisfy development goals, and (3) development goal monitoring can be used to manage inconsistencies in a developing requirements document. The application of DealScribe demonstrates that a dialog metamodel can enable a powerful environment for managing development and document inconsistencies.

This chapter expands on prior presentations of this work by focusing on implementation (Robinson and Pawlowski, 1999). Of course, the overarching theory of goal monitoring is presented. In addition, ConceptBase classes and queries that are used in DealScribe are presented.

6.1 Introduction

Requirements engineering can be characterized as an iterative process of discovery and analysis designed to produce an agreed-upon set of clear, complete, and consistent

system requirements. The process is complex and difficult to manage, involving the surfacing of stakeholder views, developing shared understanding, and building consensus. A key challenge facing the analyst is the management and analysis of a dynamic set of requirements as it evolves throughout this process. Techniques have been developed to support aspects of the process; however, support for requirements development monitoring has been lacking.

Requirements development monitoring entails (1) the specification of requirements development goals and (2) the creation of development-goal-monitoring agents. As a requirements development unfolds, monitoring agents provide warnings, and even remedies, when development goals are not satisfied. Such monitoring is similar to the performance monitoring repeatedly applied to software as part of run time optimization. However, requirements development monitoring differs in that (1) it applies over the (longer) lifetime of a requirements development and (2) the goals monitored often refer to complex interrelationships among the development products over time.

Analysts can benefit from requirements development monitoring. Since requirements development often involves changes in analysts, stakeholders, and requirements, managing a requirements development can be a challenge. Monitoring addresses this challenge by providing analysts with (1) notifications when the development does not satisfy development goals and (2) guarantees when the development does satisfy development goals.

The research discussed in this chapter defines an approach to requirements development monitoring. It offers an incremental improvement to the state of the art in process monitoring. The research shows how formal development goals can be translated into software-monitored goals. It is based on a formal dialog metamodel. Once dialog statement types and development goals are formally specified in terms of the metamodel, a dialog support system, including goal-monitoring agents, can be created automatically. A software tool is constructed to demonstrate the effectiveness of the approach. Additionally, a case study is conducted to demonstrate the practicality of this metamodeling approach to requirements development monitoring.

6.1.1 Managing Requirements Dialog

Stakeholder dialog is a pillar of the requirements development process. Techniques have been developed to facilitate dialog (e.g., Joint Application Design [JAD], prototyping, serial interviews) and to document and track requirements as they evolve (e.g., CASE). A requirements dialog can be viewed as a series of conversations among analysts, customers, and other stakeholders to develop a shared understanding and agreement on the requirements of the system under development. Typically, the analyst converses with the customers about their needs. In turn, the analyst may raise questions about the requirements, which lead to further conversations. Within

the development team, analysts will also converse among themselves about questions that have arisen during their analysis of the requirements—sometimes the result of sophisticated analysis; at other times, the result of simply reading two different paragraphs in the same requirements document.

Like many dialogs, requirements development can be difficult to manage. Empirical studies have documented the difficulties and communication breakdowns that are frequently experienced by project teams during requirements determination as group members acquire, share, and integrate project-relevant knowledge (Krasner, Curtis, and Iscoe 1987; Walz, Elam, and Curtis 1993). Requirements or their analyses may be forgotten. Different requirements concerning the same objects can arise at different times. Inconsistency, ambiguity, and incompleteness are often the result.

The objective of the research discussed here is to address the monitoring of requirements dialog goals, especially requirements consistency development goals. Requirements inconsistency is a critical driver of the requirements dialog. If one can manage requirements inconsistency, one can manage a key driver of complexity and confusion in the requirements dialog. Goal monitoring simplifies inconsistency management by alerting analysts to events important in their work. Analysts can filter the chaotic activities of a requirements dialog through goal monitors and focus their attention on important changes.

In this chapter, I present a metamodeling approach to requirements development monitoring (section 6.2). A description of a supporting software tool (section 6.3), DealScribe, illustrates how the metamodel can provide automated support for such monitoring. The requirements development protocol of section 6.4 illustrates how development goals can be expressed as instances of the dialog metamodel. That protocol, Root Requirements Management, can be monitored by DealScribe. Finally, I conclude (in section 6.5) that the dialog metamodel can provide analysts with automation, assurance, and understanding that facilitates the management of inconsistencies that arise during multistakeholder requirements dialog. The remainder of this introduction motivates the research discussed in the chapter and places it in context.

6.1.2 A Need to Support Analysts in Inconsistency Management

Requirements analysts need tools to assist them in reasoning about requirements. To some degree, computer-aided software engineering tools have been successful in providing support for modeling and code generation (Chikofsky and Rubenstein 1988; Lempp and Rudolf 1993; Norman and Nunamaker 1989); however, they have been less successful in supporting requirements analysis (Lempp and Rudolf 1993). In fact, the downstream life cycle successes of these tools may be one of the reasons that systems analysts spend a greater percentage of their time on requirements analysis than ever before (Graf and Misic 1994). Thus, analysts will benefit from the development of techniques and tools that directly address requirements analysis.

A significant part of requirements analysis concerns the identification and resolution of requirements faults. Such faults include incorrect facts, omissions, inconsistencies, and ambiguities (Meyer 1986). Many current research projects are aimed at identifying such faults in requirements. These include projects involving model checkers, terminological consistency checkers, and knowledge-based scenario checkers; additionally, more generic tools, such as simulation and visualization, are available to requirements analysts as well. For the most part, these tools are cousins of similar tools applied to programming languages that check for syntactic errors or perform checks of program inputs and path execution. However, requirements faults are rarely traced back to the original stakeholders, and there has not been much support for resolving such faults. Yet there is still a belief that conflict identification and resolution are key in systems development (Lyytinen and Hirschheim 1987; Robey, Farrow, and Franz 1989).

Empirical studies of software development projects have identified a need for issue-tracking tools (Curtis, Krasner, and Iscoe 1988; Walz et al. 1987). Typical problems of software development projects include (1) unresolved issues that do not become obvious until integration testing and (2) a tendency for conflicts to remain unresolved for a period of time. Inadequate tools for tracking issue status (e.g., conflicting, resolved) has been identified as a great concern to practicing system engineers.

Collaborative CASE tools may provide an answer. Collaborative CASE aims to support task-, team-, and group-level analysis by addressing information control, sharing, and monitoring (Vessey and Sravanapudi 1995). However, many collaborative CASE tools still fall short in their management of the requirements elicitation and development process (Liou and Chen 1993–1994). Davy (1990) notes that ''current implementations of CASE tool technology encourage an individual approach to work, even if they provide multi-user capability. However, analysts improve the quality of their work by discussing and reviewing it with colleagues. CASE tools are useful, but they are no substitute for interactive group discussion. Until more responsive interfaces are available we will only be able to make limited use of them'' (15).

6.1.3 Research Addressing Requirements Management

There is a growing literature on requirements inconsistency management. Fickas and Feather (1995) have proposed requirements monitoring to track the achievement of requirements during system execution as part of an architecture to allow the dynamic reconfiguration of component software. Feather (Cs3 2007) has produced a working system, called FLEA (Formal Language for Expressing Assumptions), that allows one to monitor events defined in a requirements monitoring language. Emmerich et al. (1999) have illustrated how the techniques used by FLEA may be used to monitor process compliance; for example, compliance with ISO 9000 or Institute of Elec-

trical and Electronics Engineers (IEEE) process descriptions (Mazza et al. 1994). The work on dialog monitoring discussed in this chapter is derived from these works, but also includes an element of dialog structuring.

Two research projects have explicitly addressed requirements dialog structures. First, Chen and Nunamaker (1991) have proposed a collaborative CASE environment, tailoring GroupSystems decision room software, to facilitate requirements development. Using collaborative CASE, one can track and develop requirements consensus. Second, Potts, Takahashi, and Antón (1994), have defined the inquiry cycle model of development to instill some order into requirements dialogs. Requirements are developed in response to discussions consisting of questions, answers, and assumptions. By tracking these dialog elements, dialog is maintained, and inconsistency, ambiguity, and incompleteness are kept in check through specific development operations (e.g., scenario analysis).

Workflow and process modeling provide some solutions for the management of requirements development (Sheth et al. 1996). It is possible, for example, to generate a work environment from a hierarchical multiagent process specification (Miller et al. 1998). There has been some attempt to incorporate workflow and process models into CASE tools (Mi and Scacchi 1992). However, these tools generally aid process enactment, through constraint enforcement. However, as Leon Osterweil and Stanley Sutton (1996) note, ''Experience in studying actual processes, and in attempting to define them, has convinced us that much of the sequencing of tasks in processes consists of reactions to contingencies, both foreseen and unexpected.''

In support of a reactionary approach, the dialog metamodel eschews process enforcement and supports the expression and monitoring of process goals. It is neutral in regard to the foregoing approaches. A methodology, conflict ontology, or automated techniques can be defined as instances of it. The DealScribe implementation provides automated support for the enactment of a structured dialog, which is defined as an instance of the dialog metamodel.

6.1.4 Support of Development Goal Monitoring

Goal monitoring addresses technical and social forces that give rise to requirements inconsistencies and conflicts by managing the changes that directly affect requirements. Changes in stakeholders, analysts, requirements, or analyses are tracked as part of the dialog metamodel. For example, a software development organization may seek an equal contribution from each of the stakeholders in a particular development effort. The dialog goal `NearlyEqualContribution` captures this:

```
NearlyEqualContribution:
"Each stakeholder must contribute a nearly equal number of
statements to the requirement dialog"
```

The `NearlyEqualContribution` goal reflects a social development goal that is monitored through the relative number of statements contributed by stakeholders.

Requirement traceability enables the monitoring of the `NearlyEqualContribution` goal. Gotel and Finkelstein (1994) define requirements traceability as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" (94). To monitor the `NearlyEqualContribution` goal, there must be backward traceability from the requirement to the contributing stakeholder. A framework that tracks development objects, and the agents and operations that act on them, supports traceability (Ramesh and Dhar 1994). Goal monitoring supports such traceability.

Goal monitoring brings traceability to life through its notifications. As development occurs, stakeholders receive feedback on interesting events; for example, warnings on the violation of the `NearlyEqualContribution` goal. Goal monitors can provide feedback on changing or inconsistent requirements which may lead analysts to become aware of additional social problems, such as conflicting stakeholder requirements.

6.1.5 Problems of Development Goal Monitoring

In defining a system that supports development goal monitoring, three main questions must be answered:

1. *How can development goals be specified?* Development goals describe relationships among the stakeholders, products, and processes of the development dialog. A goal specification language should facilitate such descriptions.

2. *How can development goals be monitored?* As the development dialog is enacted, events occurring in the dialog may result in goal failure. Goal monitoring should detect such goal violations as they occur.

3. *How can violated development goals be restored?* When a development goal is violated, a goal-monitoring system should facilitate the automated reestablishment of the goal.

The following section presents an approach that answers each of these questions.

6.2 A Dialog Support System and Metamodel

The requirements dialog support system has been designed to provide solutions to the problems of development goal monitoring, as well as to address the following basic needs of requirements dialog support:

- The need to represent multiple stakeholder requirements, even if conflicting
- The need to identify and understand requirements interactions

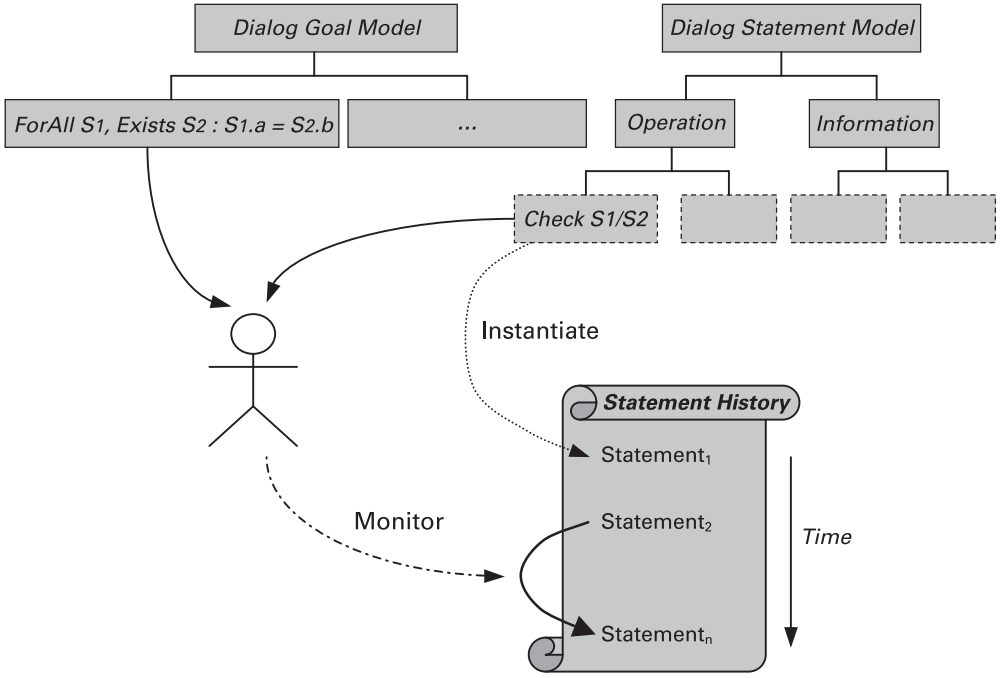


Figure 6.1
An illustration of an application of the dialog metamodel

- The need to track and report on development issues
- The need to develop shared understanding and consensus through requirements analysis and negotiation
- The need to support dynamic, dialog-driven requirements development

Stakeholder dialog is central to addressing these needs. Additionally, analysts must be able to analyze developing requirements. The needs can be supported through a dialog support system and its metamodel.

This chapter describes such a dialog support system and its dialog metamodel. As illustrated in figure 6.1, the dialog system regards a dialog as a stream of statements. Each statement is either passive information or an active operation. Statements are instances of the dialog statement model. A dialog is a continual stream of statements from multiple stakeholders; some statements initiate new ideas, and others are in response to previous statements. As the dialog expands, dialog goals can be compared against the dialog to determine their status.

The metamodel can be instantiated to define a typical process model, with a distinction between process and product. Consider information statements as products,

operation statements as actions, and dialog goals as defining a process model. The dialog metamodel is then a process model with an explicit representation of the process goals and enactment history. Such a metamodel is suitable for modeling the process of requirements development.

The phrase ''dialog metamodel'' has been chosen here, rather than the more common phrase ''process model,'' because of the specialized modeling of dialog processes and the use of metamodeling. The dialog metamodel can be instantiated to aid the contextual needs of a development group. The dialog support system provides a user interface tailored to the instantiated dialog model. As stakeholders engage in the dialog, the dialog support system provides automated processing; for example, notifying stakeholders of interesting dialog events, such as a violation of a dialog goal.

6.2.1 Dialog Support System Components

The dialog support system has four main components:

• *Dialog statement model* Statements are added to the dialog by the people, or agents, involved in the dialog. In the dialog statement model, there are two important subtrees in the statement typology:

• *Information* A passive statement that adds new information to the dialog directly or by reference to some external information source

• *Operation* An active statement that adds new information derived through some computation based on the state of the dialog as captured in the dialog forum

• *Dialog forum* The dialog forum is a statement history that includes the statements *asserted* or *retracted* as part of the dialog. Forum statements are instances of statement types that are specified in the statement typology. The statement instances include values for attributes, as well as a belief interval indicating the time at which the statement was asserted, and if it was retracted, the time of retraction. Essentially, the forum is a log of statements that have occurred during the dialog. Dialog events (i.e., assertion or retraction) may be initiated asynchronously by different stakeholders. To keep stakeholders aware of forum activities, the dialog system can notify stakeholders of new dialog events.

• *Dialog protocol* A dialog protocol is a declarative prescription of ''dialog rules,'' indicating such things as the relative order of statements and their content. In the dialog metamodel, a dialog protocol is represented by a hierarchy of dialog goals that specify desired forum properties. Examples of dialog protocols include Roberts' Rules of Order and the software development life cycle. Enforcement of the dialog protocol may be carried out through statement constraints that restrict the addition of statements to the dialog. Conversely, statements may be unrestricted, but opera-

tions can analyze the forum to determine the degree of compliance with a dialog protocol.

• *Dialog monitor* A dialog monitor is a predefined operation. After each dialog event, each believed monitor is automatically activated by the dialog system. A monitor itself specifies conditions under which another operation is to be automatically executed. For example, one predefined operation is `GoalCheck`. It determines those statements, if any, that fail to satisfy a particular dialog goal. Thus, a monitor can specify that a specific instance of `GoalCheck` is to be activated after every dialog event to maintain a report on a specific goal's status. (Predefined operations, such as `GoalCheck` and `Monitor`, are defined in the same manner as user-defined operations.)

6.2.2 Dialog Metamodel Overview

The main components of the dialog support system are defined in the dialog metamodel, as illustrated in figure 6.2. This metamodel defines the generic system classes. Before the system can be applied, classes of the metamodel must be specialized for a specific dialog context. For example, in the context of requirements development, a statement type, `Requirement`, may be made an `isA` subclass of the `Information` statement class. Such specialized classes define a context-specific dialog model. During the application of the dialog system, instances of `Requirement` may be asserted as part of a requirements dialog. These dialog model instances capture the content of the stakeholder dialog.

The dialog metamodel is illustrated in figure 6.2 as an entity-relationship diagram. It shows how a dialog `Forum` consists of a set of dialog `Statements`. Each `Statement` instance must be an instance of the `Operation` or `Information` class or an instance of their (user-defined) subclasses.

The `GoalCheck` operation is a predefined subclass of `Operation`. An instance of `GoalCheck` specifies an instance of a `Goal` that is to be checked when the `Goal` is activated. (A `Goal` itself may be defined as an AND/OR tree of goals.) A `Goal` is checked by determining whether its properties hold true within a given `Forum`. Each `Goal Property` is a logical expression that refers to dialog model instances of a specific `Forum`. When specific statements can be identified as failing to hold for a particular goal, their failures are associated with that goal.

As illustrated in figure 6.2, `Monitor` is also a predefined subclass of `Operation`. It is automatically activated when the properties of its trigger hold for a specific `Forum`. Once activated, it may activate other operations.

A `Practice` names a set of properties and may be justified by `Rationale`. For example, defining user priorities for system requirements may be a practice of an incremental development protocol (i.e., a standard [Mazza et al. 1994]) justified by the

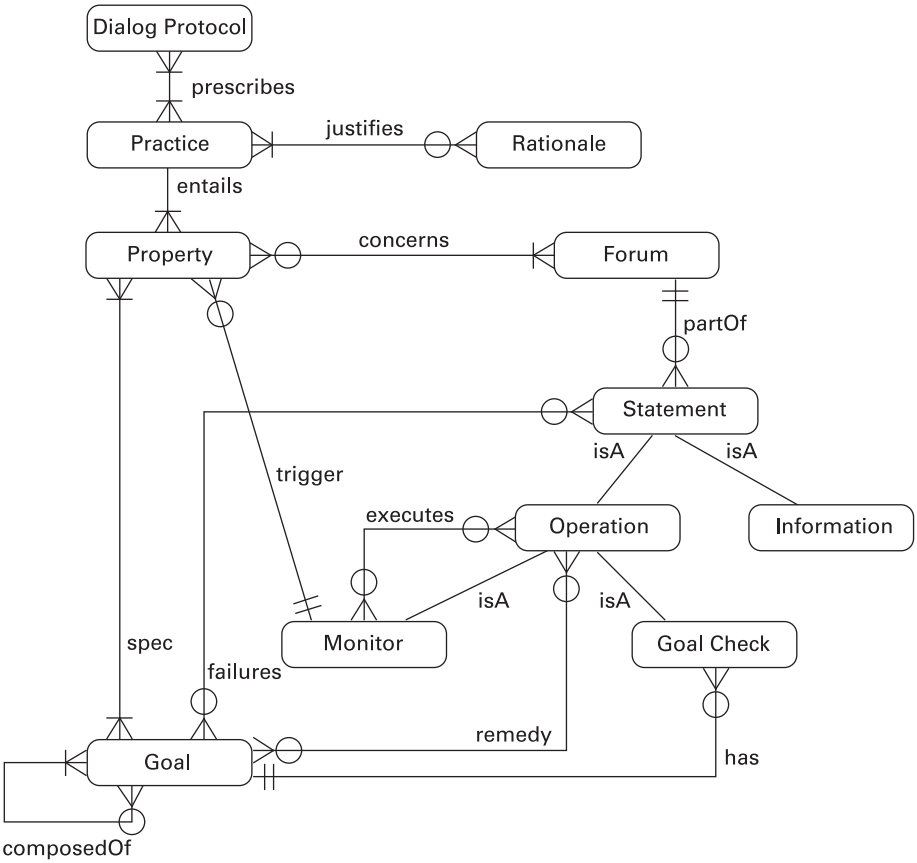


Figure 6.2
An entity-relationship diagram of the main dialog metamodel entities

use of those priorities in scheduling the incremental delivery of a system. The practice of defining user priorities for system requirements may be specified using logical properties of the `Forum`:

```
HaveUserPriority:
∀ R ∈ Requirement, ∃ F ∈ Forum, ∃ P ∈ UserPriority
• (In R F) ∧ (HasPriority R P)
```

According to the definition of `HaveUserPriority`, all `Requirement` statements must have an associated user priority. A set of such `Practices` defines a `Dialog Protocol`.

Although all entity instances of the dialog metamodel may be defined as logical expressions, only the properties of goals and monitors require this type of definition. In general, the entities `Dialog Protocol`, `Practice`, and `Rationale` may be informal text. A dialog protocol is simply a name given to a set of practices. It is made operational by formally defining and composing dialog goals that represent the practices of the protocol.

6.2.3 Dialog Statements

Dialog statements are defined as standard object classes with constraints according to the following template:

```
Class <statement-name> isA <Type> [, <Type>]* with
  attribute [ <attribute-name> : <Type> ; ]*
  constraint [ <constraint-name> : <constraint-expression> ; ]*
end
```

Statement types are organized into an inheritance (`isA`) typology. A user may attempt to add a statement to a forum by assigning attribute values to a statement type, thereby creating a statement instance. If the constraint expression does not evaluate as FALSE, then the statement is indeed asserted to the forum. (Constraints are logical expressions, as illustrated in the following subsection.) Operational statements do more than simply assert attribute values. An assertion of an operational statement causes the execution of its associated (`onAssert`) method. Such methods may engage in computation and assign statement attribute values. Finally, the resulting statement is asserted to the forum.

6.2.4 Dialog Goals

As `HaveUserPriority`, defined previously, illustrates, dialog goals may be specified in terms of forum properties. Often a goal specification, like that of `HaveUserPriority`, concerns properties of a certain statement type. In such cases, the dialog system can determine the specific statements that fail the goal (indicated as failures in figure 6.2). However, a goal specification may concern summary properties across sets of statements, such as the number of high-priority requirements:

```
LessThan15HighPriorityRequirements:
∀ R ∈ HighPriorityRequirement,
  ∃ F ∈ Forum, ∃ N ∈ Integer
• (In R F) ∧ (Count R N) ∧ (N < 15)
```

The goal specification for `LessThan15HighPriorityRequirements` is met for a particular goal when there are less than 15 high-priority requirements for the goal.

However, there are no particular individual statements that will fail the goal specification. (One might consider the fifteenth high-priority requirement asserted as the statement that caused the goal failure; however, this captures only part of the goal's intent.)

In general, a dialog goal expresses arbitrary logical formulas concerning a forum. A change in any value within the scope of a formula can cause its evaluation to become FALSE, thereby making failure attribution difficult. For example, a goal concerning a deadline may fail with the passage of time. Nevertheless, the dialog system can determine whether such a goal specification is met in a forum. Moreover, specific statements that fail a goal can be determined for dialog goals that characterize properties of a specific statement type.

Goals also have an intentional mode that is used in combination with the goal specification. A goal mode may be `Achieve` or `Avoid`. The goal mode alters the way in which satisfaction of the goal specification is interpreted. In `Achieve` mode, if the goal specification is met, then the goal succeeds. Conversely, in `Avoid` mode, if the goal specification is met, the goal fails. (This leads to the corollary, `Achieve(g)` = `Avoid(¬g)`.)

A third goal mode, `Maintain`, is provided for convenience. A goal with mode `Maintain` that is monitored prevents the assertion of any statement that violates the goal's specification. Thus, a single goal in `Maintain` mode can be used in lieu of assertion constraints in multiple statement types. The statement `Maintain(g)` is similar to `Achieve(g)`; however, if `Maintain(g)` is monitored, then statements cannot be asserted as the goal fails.

The two dimensions of goal mode and specification provide a simple means of altering the interpretation of goals. For example, in the early phase of a project, one might apply `HaveUserPriority` as an `Achieve` goal. During this phase, requirement statements may be asserted without user priorities; each such requirement causes the failure of `HaveUserPriority`. Such failures typically lead to user notification and eventual statement modifications. During the final phase of the project, all requirements will have priorities. To prevent the addition of new requirements that do not have a user priority, the mode of `HaveUserPriority` is set to `Maintain`.

As illustrated in figure 6.2, a goal may also specify remedy operations. Such operations may be applied in the case of the goal's failure to reestablish the goal's satisfaction. `GoalCheck` is an operation that is used to activate goal remedies. When executed, it checks whether the specified goal has failed. If so, and a Boolean `runRemedy` flag is set to TRUE, then the remedies are executed.

Many goal failures may occur as the result of the same dialog event. However, each remedy operation is executed in response to a single goal failure. Thus, there is no predefined global analysis of all goal failures followed by a global remedy. There are two means of addressing this problem.

First, a metagoal concerning multigoal failure can be asserted. For example, a slight modification of the `LessThan15HighPriorityRequirements` goal yields the goal `MetaGoal-LessThan3ActiveFailedGoals`:

```
MetaGoal-LessThan3ActiveFailedGoals:
∀ G ∈ ActiveFailedGoal, ∃ F ∈ Forum, ∃ N ∈ Integer
• (In G F) ∧ (Count G N) ∧ (N < 3)
```

This goal fails if more than three active goals fail. The goal subclass `Active-FailedGoal` contains those goals that are currently being monitored and have failed. A remedy for `MetaGoal-LessThan3ActiveFailedGoals` can conduct global analysis of the failed goals in order to provide a more global remedy than remedies for single goal failures. In general, as an operational statement, a remedy operation may do complex programming activities, possibly involving user input, to modify the forum; for example, to modify all statements that have led to a goal's failure.

A second means of remedying the lack of a predifined global goal failure analysis concerns composite goals. A composite goal specifies an AND/OR goal tree. A composite goal fails if one of its AND goals fails or if all of its OR goals fail. When `GoalCheck` is applied to a composite goal, remedies at each level of the goal tree are applied to the failed subgoals. (The implementation follows a planning paradigm, with remedies for the leaf goals executing first, followed by those for higher subgoals, and ending with the topmost goal remedies.) Thus, the goal tree is used to specify goals, recognize their failure, and apply remedies to bring about goal success (cf. Klein 1991).

6.2.5 Dialog Protocols

Dialog goals are used to operationalize a dialog protocol. The practices of a dialog protocol are specified in the formal properties of dialog goals. Dialog goals themselves may be organized in an AND/OR goal tree. Thus, a dialog protocol is formalized as a composite dialog goal. A dialog protocol is activated (or inactivated) through monitoring operations.[1]

6.2.6 Monitoring

`Monitor` is a predefined operation, as illustrated in figure 6.2. A monitor has a trigger that specifies properties that must hold for the monitor to be active. A monitor also specifies operations that are to be executed when the monitor is active. After each dialog event, the operations of each active monitor are executed.

Monitoring can be used to automate two types of tasks. First, basic operations can be automatically run. This includes keeping analysis current and automating synthesis. Second, goals can be monitored to provide alerts (or remedies) when goals fail.

Commonly, a monitor specifies that an operation is to be executed after every dialog event. For example, a specific operational statement, such as `GoalCheck(HaveUserPriority)`, can be monitored to ensure that a goal failure of `HaveUserPriority` is immediately detected and possibly remedied.[2] However, some operations are computationally expensive; for example, checking goal failure in a large goal tree or applying a complex remedy. In such cases, the monitor can be used to selectively invoke the operation. Monitors may be run periodically; for example, modulo the forum event count or chronological time.

Monitoring may inadvertently introduce recursion among operations. After each dialog event, each monitor is activated. As a result, remedy operations may be executed and their results asserted as statements. The new statements may, in turn, activate more remedy operations that lead to more statements, and so on. As a pathological example, consider a goal `HaveResponse` that specifies that all statements must have a response:

```
HaveResponse:
∀ S ∈ Statement, ∃ F ∈ Forum, ∃ R ∈ Statement
• (In S F) ∧ (Response S R)
```

A remedy operation for `HaveResponse` could add a new statement, S_r, as a response. Of course, S_r will need a response as well. In general, such recursion must be controlled through careful definition of goals and remedy operations.[3]

As a final point on monitoring, notice that monitoring of monitors follows naturally from the framework. For example, it can be specified that the goal `HaveUserPriority` should be periodically checked by `GoalCheck`:

```
MonitorPriority:
∀ M ∈ Monitor, ∃ F ∈ Forum, ∃ GC ∈ GoalCheck
• (In GC F) ∧ (GC Goal HaveUserPriority) ∧ (In M F) ∧ (M Operator
  GC) ∧ (M EventPeriod 5)
```

The goal `MonitorPriority` is met when the goal `HaveUserPriority` is checked periodically by a monitor, `M`. (`M` executes `GoalCheck` on the `HaveUserPriority` goal.) Of course, since `MonitorPriority` is a goal, it can also be monitored.

6.2.7 Hypothetical Statements

With statement constraints, goal specifications and their remedies, and operation monitoring, the consequences of adding a statement to the dialog can be difficult to predict. To facilitate user understanding, the dialog system provides an assertion mode for statements. In standard mode, a statement is asserted, followed by the activation of monitors and their operations. In hypothetical mode, a statement is tenta-

tively asserted, and the subsequent events of monitors and operations are displayed but not asserted. Such a mode allows users to see the consequences of adding a statement to the dialog without actually asserting it.[4]

6.3 Tool Support for the Dialog Metamodel

The dialog metamodel and support system form a conceptual design for the support of requirements analysis dialogs. To validate the effectiveness of the design, a software implementation was constructed. The implementation environment supports not only the current design, but the iterative refinement of the design as well.

The dialog support system implementation is called DealScribe. It is part of the DealMaker tool suite aimed at supporting requirements negotiation (Robinson and Volkov 1998).

6.3.1 Dialog System Architecture

The three main components of DealScribe interact over a network interface:[5]

• *Database server* The database server stores dialog messages, checks constraints and rejects messages that violate them, answers queries concerning goal status and active monitors, and answers hypothetical queries. The deductive database ConceptBase provides these functions. It provides concurrent multiuser access to O-Telos objects (Jarke et al. 1995). All classes, metaclasses, instances, attributes, rules, constraints, and queries are uniformly represented as objects (Mylopoulos et al. 1990). ConceptBase provides a powerful operational modeling language based on Datalog with negation (Ceri, Gottlob, and Tanca 1990; Minker 1996); it will terminate and produce the correct answer for any query it receives. Thus, one can formally describe a model in ConceptBase, populate it with instances, and have ConceptBase answer queries about the model or instances of it.

• *User interface server* The user interface server provides message input forms, dialog forum and message views, multiple forums, e-mail notification of new messages, and secure administration of user access. A modified version of the World Wide Web discussion system HyperNews provides these functions. In the modified version, a user can post *typed* messages to forums, with the types defined in the database server. A view of the forum can provide an overview of the discussion, in which messages are laid out in a tree format that shows replies to a particular message indented under it (see figure 6.6). Statements are stored in ConceptBase and as Web pages. Thus, one can run both ConceptBase (logical) queries and Web search engine (match-based) queries, such as those provided by Excite. Finally, Web collaborative tools (e.g., whiteboard) can be invoked from DealScribe, as an operation, and their content stored in a forum.

• *Clients* A Web browser serves as the basic user client. Using this network interface, multiple clients can interact with DealScribe from any Internet connection. Additional client interfaces include a graphical browser and a (Perl) program Application Program Interface (API).

In the following subsections, I summarize how the dialog system functions are implemented.

6.3.2 Dialog Statements

The statement template presented in section 6.2.3 is essentially the syntax of ConceptBase class definitions. Thus, a DealScribe information statement, such as `Requirement`, can be defined similarly. For example, `Requirement`, with attributes `perspective`, `mode`, `description`, and `contention`, can be defined as follows:

```
Class Information isA DealScribeMessage
end

Class Requirement isA Information with
  attribute
    perspective : Perspective;
    mode : Mode;
    description : String;
    contention : FuzzyNumber
end
```

According to this definition, a `Requirement` is a type of `Information` class. The `Information` class is a type of `DealScribeMessage` class. These ConceptBase defintions correspond directly to the entities illustrated in figure 6.2. However, in this implementation, the `DealScribeMessage` class represents the `Statement` class illustrated in figure 6.2.

From such ConceptBase definitions, DealScribe can directly generate HTML input forms. A user can fill in, or select, values for attributes of the statement object. Operation statements are similarly defined. For example, `GoalCheck` is defined as follows:

```
Class Operation isA DealScribeMessage with
  attribute
    resultString : String;
    result : Proposition
end

Class GoalAnalysis isA Operation
end
```

```
Class GoalCheck isA GoalAnalysis with
  attribute
    goal : DialogGoal;
    runRemedy : Boolean
end
```

Like information statements, the object attributes of operation statements may serve as input fields for HTML forms; similarly, other attributes may serve as output fields.

As illustrated in figure 6.2, both informational and operational statements are subtypes of `Statement` and are part of a `Forum`. This is implemented in ConceptBase by making all statements a subtype of the `DealScribeMessage` class:

```
Class DealScribeMessage with
  attribute
    dsTransactionNumber : Integer;
    datetimestamp : Integer;
    date : String;
    title : String;
    userid : String;
    username : String;
    keywords : String;
    body : String;
    msgtype : String;
    replyto : String
end
```

The `DealScribeMessage` class captures attributes common to all HyperNews messages. Thus, the DEALSCRIBE system maintains a representation of all messages found in a HyperNews (HTML) forum within a ConceptBase database.

Operational statements, such as `GoalCheck`, discussed previously, have an associated method in addition to their basic ConceptBase class defintion. Currently, all such methods are defined in the Perl programming language. When an operation is executed, its ConceptBase results are stored in the `result` attribute, and an ASCII string representation of those results is stored in the `resultString` attribute.

The following presents a portion of `GoalCheck`'s associated Perl subroutine. The Perl `GoalCheck` subroutine is called when it is time to check a goal. The name of a specific goal is passed to it as an argument. The Perl subroutine shown here connects to the ConceptBase database to determine the goal's mode, as well as to determine those statements that have violated the goal:

```
sub GoalCheck {
  my($AttAssoc,$in) = @_;
  # Retrieve the goal name from the HyperNews message.
  my $goalName = ${$AttAssoc}{"goalName"};
  # Determine the goal's mode (using ConceptBase).
  my $checkMode = CBrun('ask_objnames', "checkMode[$goalName/
  goal]");
  # Ask ConceptBase to determine
  # the Violations of the goal.
  my $violationMsgs =
  GoalFailureMessages($goalName,"Violation",$checkMode);
  my $resultString = "";
  my $msg;
  # Place the failed messages names
  # in the resultString ...
  for $msg (@$violationMsgs) {
    $resultString .= "$msg->{name} \n";
  }
  # Set the result in the HyperNews message.
  ${$AttAssoc}{resultString} = "\"$resultString\"";
  # ... more code (elided)
}
```

Figure 6.3 shows a portion of a `GoalCheck` statement from DEALSCRIBE. As a result of the user's selecting a goal and then posting the operation, `GoalCheck` found statements that failed the goal and presented them as Web links.

Figure 6.4 shows a portion of a DEALSCRIBE Web page for adding a statement. The table of statement types depicted in the figure and the input and output forms associated with them are automatically generated from the corresponding ConceptBase statement model. The ConceptBase statement hierarchy is depicted in figure 6.5. A user can post a statement by selecting a statement type and then clicking the "Add Message" button. All aspects of the user interface that are dependent on the metamodel are automatically generated. Thus, modifying a dialog model, even dynamically, is easy. A new statement type is simply defined, and the statement buttons and input-output (I/O) forms are then automatically updated.

6.3.3 ConceptBase Statement and Query Semantics

ConceptBase classes, like the aforementioned `GoalCheck`, define data types that include typed attributes and constraints. These data types and their instances are stored in a deductive database. As `GoalCheck` illustrates, ConceptBase is an object-oriented

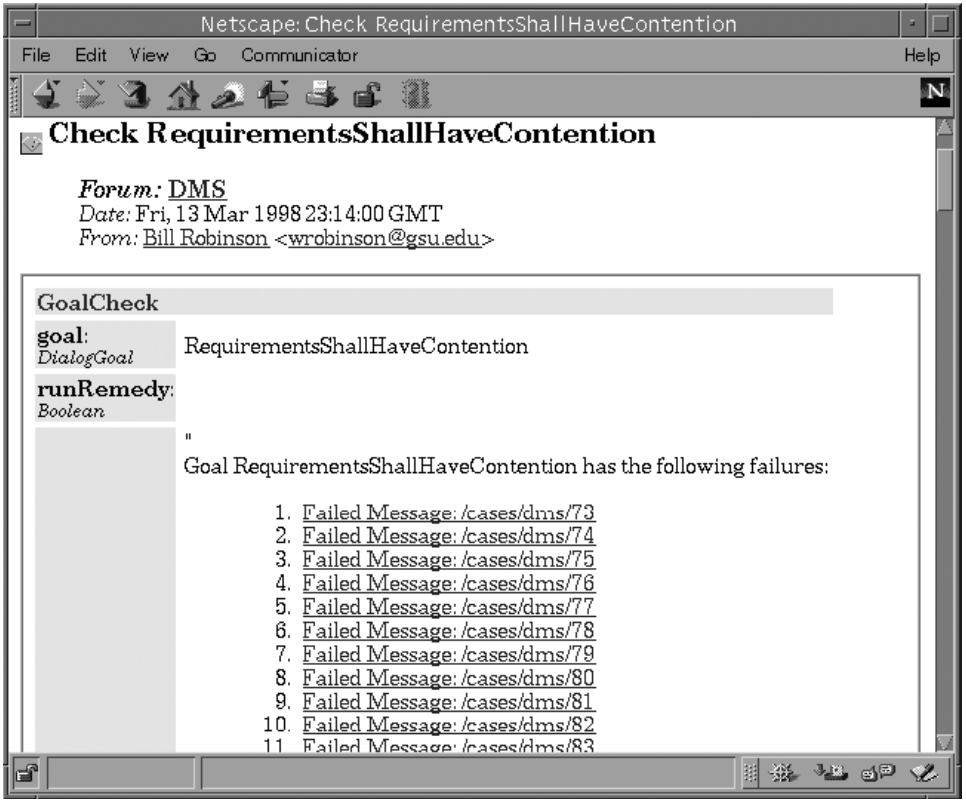


Figure 6.3
A portion of a DEALSCRIBE Web page showing the results of asserting a `GoalCheck` statement. The page is representative of the input and output forms. Each object attribute is depicted as a row heading in a table. Attribute values are depicted as the contents of table cells.

database. Moreover, it uniformly integrates concepts of deductive databases. Without leaving the Datalog (with negation) framework, it makes object-oriented abstraction mechanisms available to the user, thus providing significant help for data structuring (as compared with that offered by relational deductive databases). The query language supported is similar to those offered by other object-oriented databases but, as in deductive databases, is more directly integrated with the rest of the data model; this has led to some useful ideas with many applications, such as the parameterization of query classes (Jarke et al. 1995).

ConceptBase provides deductive database techniques, such as arbitrary relationships between tables, recursive processing of tables, and parameterized query classes. Although such techniques are being incorporated into commercial databases, most

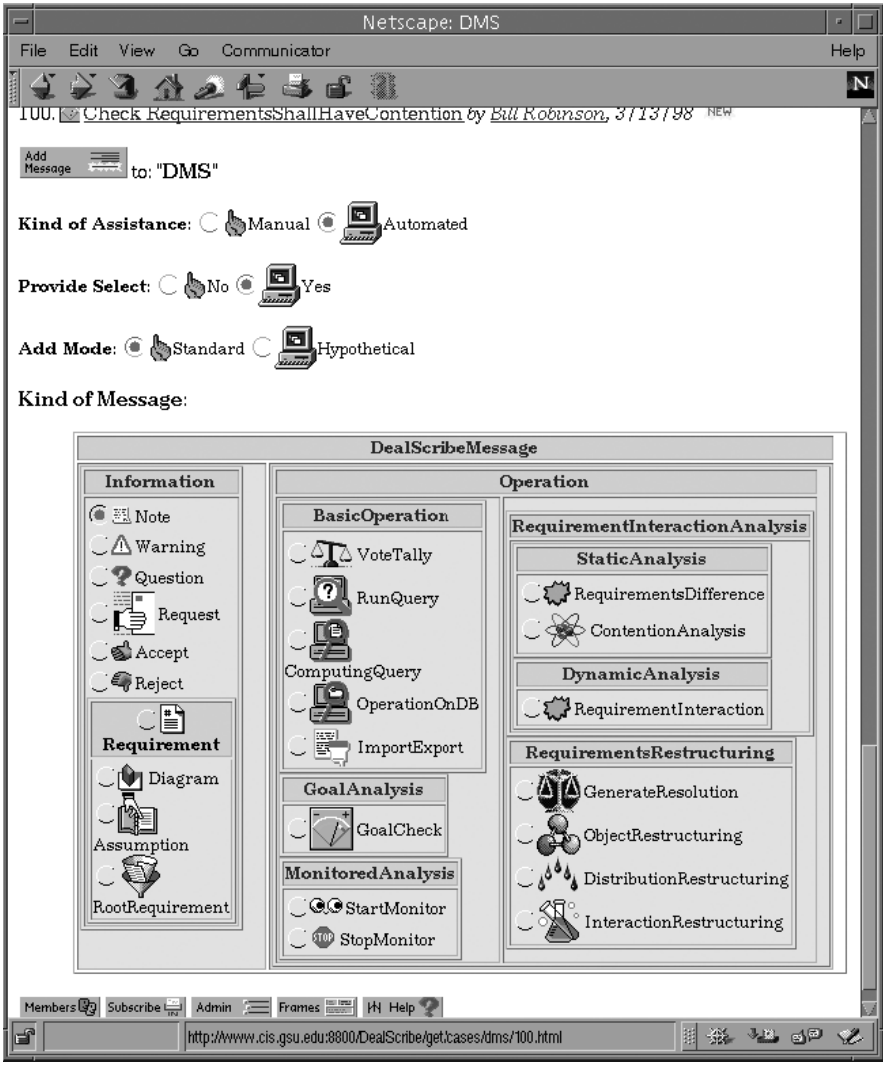


Figure 6.4
A portion of a DEALSCRIBE Web page showing the selection of statement types as a hierarchy of radio button tables. Each type is defined in the ConceptBase statement model.

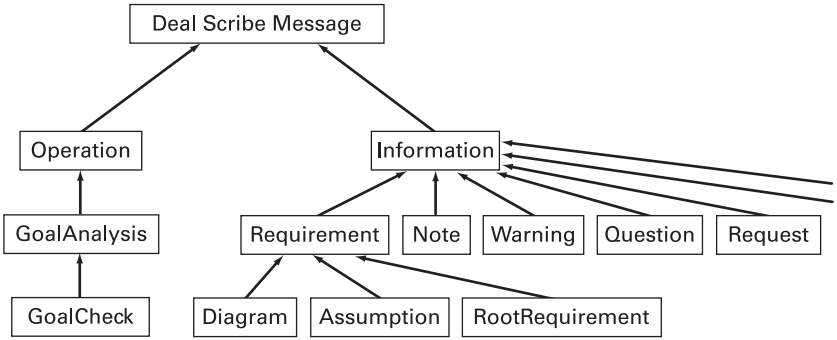


Figure 6.5
A portion of the ConceptBase database showing part of the hierarchal statement model

commercial systems still lack these features (Minker 1996). Nevertheless, these features have been helpful in the development of DEALSCRIBE—particularly, the parameterized query class.

ConceptBase query classes are defined in the same way as its data classes. A QueryClass is itself an instance of one or more classes. Through its `constraint` and `isA` specification, a QueryClass defines necessary and sufficient conditions for objects that are instances of it. Such conditions are used to compute the objects that answer the query.

Consider the following `HaveUserPriority` QueryClass, an instance of the `DialogGoal` class:

```
QueryClass HaveUserPriority in DialogGoal isA Requirement with
mode
  mode_a : Achieve
constraint
  ID_Priority :
    $ exists x/Priority (this userPriority x) $
end
```

Objects that are members of (i.e., in) `HaveUserPriority` are constrained to be `isA` the `Requirement` class and have a value for `attribute` named `userPriority`.[6] Thus, query definitions may themselves be instances of one class, while having objects fulfilling their query constraints from another class. (This is not unlike the situation for a standard SQL query; however, typically, a SQL query's type cannot be queried.)

QueryClass definitions can be queried and manipulated in ConceptBase in the same way that any other class can be queried and manipulated. Such uniformity of

manipulation provides a concise way to define and check dialog goals. For example, one can define an operation that queries goals of the current dialog protocol as part of proving interesting properties, such as the goals' logical consistency. However, such metagoal analysis, beyond goal violation checking, is not part of DEALSCRIBE.

6.3.4 Dialog Goals

The `DialogGoal` class is defined in a way similar to statement types:

```
Class DialogGoal with
  attribute
    mode : CheckMode;
    remedy : DealScribeOperation;
    andGoals : DialogGoal;
    orGoals : DialogGoal
end
```

As illustrated previously, a specific goal is defined as an instance of this type. Goal failure can be determined through a database query. Consider the achievement of the `HaveUserPriority` goal. If a `Requirement` statement does not have a `userPriority`, then the goal fails. More generally, if a goal mode is `Achieve`, then statements in the `isA` class specified that do not satisfy the goal's constraint are statements that fail the goal. This is captured in the following rule:

```
Failure Retrieval Rule:
For a goal G with isA type T and with constraint C, goal failures
are in type T that satisfy not C.
```

Since a DEALSCRIBE goal is itself a query, it can be used to find those statements that are instances of the goal type but are not instances of the goal query. (If the mode of a goal is `Avoid`, then it finds statements that do satisfy the goal query.) The following query implements goal failure detection for dialog statements:

```
GenericQueryClass FailuresOfQueryGoal isA DealScribeMessage with
  parameter
    goal : DialogGoal;
    mode : CheckMode
  constraint
    c : $ (FALSE in InSubClassOf[goal/object,ComparisonGoalType/
    class]) and ((mode == Avoid) and (exists parent1/
    OneLinkIsA[goal/child] (this in parent1) and not (this in
    QueryClass) and not (this in goal))) or ((mode == Achieve) and
```

```
    (exists parent2/OneLinkIsA[goal/child] (this in parent2) and
    not (this in QueryClass) and (this in goal))) $
end
```

`FailuresOfQueryGoal` is a parameterized query that, given a goal and a mode, will return those statements that satisfy the constraint. `FailuresOfQueryGoal`'s constraint in turn implements the Failure Retrieval Rule previously specified. (`OneLinkIsA` returns the direct parent `isA` types of the goal.)

Notice that `FailuresOfQueryGoal` must first check that the goal does not involve arithmetic computation (for example, counting a number of class instances and then comparing that with a value). Such `ComputingGoals` must be analyzed separately, because such computations are not intrinsic to ConceptBase.

Goals whose properties cannot be directly computed in ConceptBase are analyzed separately. For example, `LessThan15HighPriorityRequirements` is an instance of `CountingGoal`. Its attributes include a `query` that is used to accumulate the items to be counted and a `goal_count` indicating a value to be compared using the indicated relation:

```
QueryClass LessThan15HighPriorityRequirements in CountingGoal with
  mode
    mode_a : Achieve
  query
    q : HighPriorityRequirement
  goal_count
    gc : 15
  relation
    c : LessThan
end
```

`CountingGoal` is a subtype of `ComputingGoal`. These goal types require some algorithmic computation that cannot be expressed in ConceptBase queries. To accommodate such goals, DealScribe computes their values externally and caches them with the ConceptBase goal. Such caching is efficient, as only those goals that are currently being analyzed will be updated.

Identifying what goals are active at any given point is simple, as the following query shows:

```
QueryClass ActiveGoalRoots isA DialogGoal with
  constraint
    c : $ exists mon/StartMonitor (mon in ActiveMonitors) and
    exists gc/GoalCheck (gc in mon.Message) and (this in gc.goal)
    $
end
```

The query `ActiveGoalRoots` returns the names of those goals that are currently being monitored. The underlying objects include instances of `ActiveMonitors` and `GoalCheck`. These objects are asserted or retracted as users add `StartMonitor` and `StopMonitor` messages to the DEALSCRIBE forum.

In general, failure for an AND/OR goal tree is implemented with a recursive query that descends the tree. The following query, `FailureMessagesFromGoalTree`, implements this descent:

```
GenericQueryClass FailureMessagesFromGoalTree isA
DealScribeMessage with
  parameter
    goal : DialogGoal
  constraint
    c : $ (this in FailsGoalMode[goal/goal]) or
    (exists ag/goal.andGoals
      (this in FailureMessagesFromGoalTree[ag/goal])) or
    (forall og/goal.orGoals
      (this in FailureMessagesFromGoalTree[og/goal])) $
end
```

When this query is applied as part of the `GoalCheck`, the user is presented with a goal tree that includes Web links to the messages that fail each goal (see figure 6.3). If the user chooses to apply the associated goal remedies, they are applied, in order, from the leaves to the root of the goal tree.

6.3.5 Monitoring

Monitors are defined in the same way as all other statement types. The following ConceptBase class defines `StartMonitor`, which is used to define a monitor instance. (A similar statement is used to stop a monitor.)

```
Class StartMonitor isA MonitoredAnalysis with
  attribute
    Trigger : Query;
    StartTime : DateTime;
    EndTime : DateTime;
    TransactionInterval : Integer;
    TimeInterval : Integer;
    Operator : DealScribeOperation
end
```

To use `StartMonitor`, a user selects an operation to be monitored and a trigger that, when non-nil, will result in the execution of the operation. Although a trigger

may involve complex temporal expressions, most triggers simply define start and end times or transaction intervals. For convenience, start and end times may be input directly.

Monitored operations are executed after every dialog event. To make this happen, the `ActiveMonitors` query retrieves instances of `StartMonitor` whose triggers are satisfied. Next, DealScribe executes each associated operation, and a record of its execution is inserted into the dialog forum. As with all statements, an e-mail message is also sent to users who have indicated that they desire e-mail notification of monitored operations.

Any operation statement can be monitored. To monitor an operation statement, (1) a user asserts an operation statement, `O`, then (2) the same user or another user asserts a `StartMonitor` statement as a response to `O`. The original assertion of `O` allows for the input parameters of `O` to be assigned. The assertion of the `StartMonitor` defines the conditions under which operation `O` will be executed. DealScribe runs the operation, according to the monitor parameters, until a `StopMonitor` is asserted for `O`. The forum, as depicted by DealScribe in figure 6.6, indicates (1) the initial assertion of `GoalCheck`, (2) the subsequent `StartMonitor`, (3) the subsequent execution of the monitored `GoalCheck`, and finally, (4) the `StopMonitor` statement. Thus, monitoring is divided into two parts: (1) the condition under which the operation will be executed and (2) the operation itself. Additionally, the operation may impose its own conditions that must be met before results are asserted.

Figure 6.6 shows a portion of a DealScribe forum Web page. The initial assertion of the `GoalCheck(RootRequirementsManagement)` operation is shown at the top of the page, and the `StartMonitor` and subsequent responses are shown below. As a result of applying `GoalCheck(RootRequirementsManagement)` early in the requirements development process, a remedy in the `RootRequirementsManagement` tree, `ContentionAnalysis`, has been automatically applied. Thus, both the monitored operation, `GoalCheck(RootRequirementsManagement)`, and its applicable remedies are executed as the result of monitoring.

6.3.6 Hypothetical Statements

Although the dialog metamodel provides for the expression and monitoring of complex interactions, the consequences of asserting any one statement can be difficult to determine. Of course, the consequences of a statement can be determined by simply asserting it. However, that may lead to a number of unwanted warning messages or remedy operations. In cases in which the user would like to explore hypothetical statements, DealScribe does not assert a statement but instead tentatively asserts the statement and then queries about the resulting state.

DealScribe uses TELL-HYPO to hypothetically run a DealScribe dialog event. In standard mode, DealScribe executes the ConceptBase TELL function to add

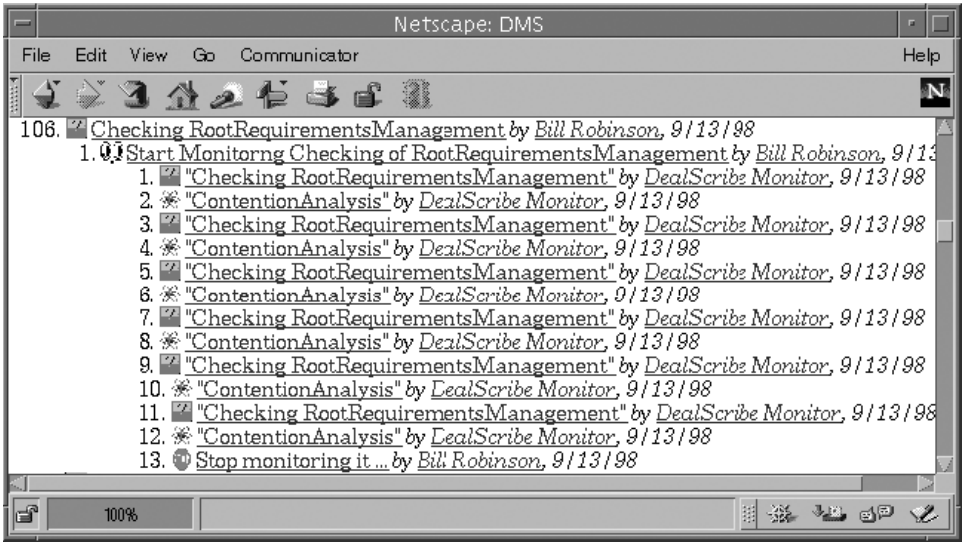


Figure 6.6
A portion of a DEALSCRIBE Web page showing statement headings (with elements in the following order: number, icon, title, author, date). The initial `GoalCheck` statement is at the top, followed by a `StartMonitor` response and the subsequent monitored responses of three types: `GoalCheck`, `ContentionAnalysis`, and `StopMonitor`. Note that `ContentionAnalysis` is the remedy of a subgoal of the monitored `RootRequirementsManagement` goal. The final `StopMonitor` response ends this monitoring of `GoalCheck`. (Responses to a message are shown indented, below the message, and with the newest statements toward the bottom.)

statements to the database. However, in hypothetical mode, DEALSCRIBE executes the ConceptBase TELL-HYPO function to temporarily add statements to the database, check constraints, and answer queries. Thus, DEALSCRIBE hypothetically asserts the statement, queries to find the hypothetical `ActiveMonitors`, and determines the consequences of the monitored operations.

Consider the standard assertion of a requirement without an assigned user priority. If the property that a requirement has an assigned user priority were being monitored, say, through `GoalCheck(HaveUserPriority)`, then a warning of goal failure would be asserted. In asserting the same requirement hypothetically, however, the same goal failure would be noted, but no actual warning message would be asserted.

6.3.7 Related Systems

The DEALSCRIBE implementation simply demonstrates one means of supporting the functionality of the dialog metamodel and system. Other support environments (e.g., DOORS (Telelogic 2007), Requirements and Traceability Management (RTM),

and RDD-100 [Alford 1992]) were considered but the decision was made to create DealScribe instead. Adapting a commercial tool to support the dialog metamodel might have created a more usable system. However, the goal was to develop an environment for iterative development of and experimentation with dialog metamodels and supporting tools. Consequently, DealScribe was built on a metamodeling tool that provided flexibility in prototyping.

Current requirements engineering tools do not include process model support for the goal tree and monitor specified in the dialog metamodel and system. However, many do have a means of adding tool extensions (i.e., an API) by which process support can be provided. For example, a similar model of standard compliance checking has been built based on DOORS (Emmerich et al. 1999).

The exploratory research presented in this chapter required a flexible metamodeling environment. Most requirements engineering tools use a relational database to support their analyses. Although this approach has advantages, the traditional relational database implementation makes metamodeling difficult. In a relational database, queries apply to data instances (records) but not to data schema. Such schema form the metamodel of figure 6.2. In contrast, both schemas and their instances are stored and queried in ConceptBase.

The metamodeling environment of ConceptBase has aided the iterative and experimental development of the dialog metamodel and support system. By querying the metamodel, external tools can adapt themselves to changes in the metamodel. For example, DealScribe input and output forms are automatically updated with changes to the metamodel of figure 6.2. Query classes also provide for analyses of model instances that are themselves schemas. For example, dialog goals are themselves queries; yet queries such as `FailuresOfQueryGoal` can be formulated to analyze dialog goals. These metamodeling features have simplified the development of DealScribe. Although such complexity may not be necessary for a commercial requirements engineering tool, it has aided the development and experimentation of the dialog metamodel.

Few commercial requirements engineering tools have a language or support environment based on typed predicate logic expressions. Queries and constraints in most tools are based on SQL or simple object model constraints. In contrast, ConceptBase provides an efficient implementation of Datalog with negation. This language simplifies the transition from the formal theoretical model to the running implementation, as illustrated in this section and the previous one.

Some research environments may be adapted to support the dialog metamodel and system. The most obvious is the DOORS requirements tool, extended to support compliance management (Emmerich et al. 1999). The extended environment can monitor the compliance of a requirements document with standard documentation

practices. Each practice references document properties as represented in the underlying DOORS database. Then an external monitor, FLEA (Cs3 2007), issues notifications when a practice is not met. DealScribe has similar capabilities; however, the two systems also have several differences:

1. DealScribe's goal language is based on the logically expressive Datalog (Minker 1996).
2. DealScribe goals can refer to operation properties as well as document properties.
3. DealScribe explicitly includes remedy operations that can reestablish goals.
4. DealScribe provides hypothetical statements.
5. DealScribe is implemented in a metamodel environment, thereby facilitating experimentation and extension.

WinWin is a commonly cited research project on requirements development groupware (Boehm and In 1996; Egyed and Boehm 1996). The WinWin tool provides groupware support for tracking team development of requirements, including conflict detection and resolution. WinWin notifies analysts when new requirements may conflict with existing requirements, according to its hierarchy of requirements conflict criteria. In this sense, it provides monitoring and notification. However, the criteria for notification are encoded into system procedures, thereby making formalism, dynamic criteria changes, or dynamic disabling and enabling of monitors difficult. Finally, the groupware aspect of the system is based on a paradigm of message exchanges, as opposed to a globally viewable message forum.

6.4 Development Goals for Managing Inconsistency

A dialog support system like DealScribe can be applied to a variety of tasks that require the management of a requirements dialog. In general, development goals that can be expressed in terms of logical expressions over informational and operational dialog statements can be monitored and maintained. For example, DealScribe goals can represent aspects of standardized processes, such as ISO 9000 or IEEE process descriptions (Mazza et al. 1994), and can execute warnings in response to goal failures (cf. Emmerich et al. 1999). In addition to issuing standards compliance warnings, DealScribe can take an active role in executing tasks. Goals can specify remedy operations that can be applied in the case of goal failure. Such goals, in combination with monitoring, can automate tasks such as synthesizing resolutions as requirements conflicts arise, totaling stakeholder votes for alternative requirements in cases where stakeholders disagree about requirements, or updating the status of requirements as beliefs concerning dependent assumptions change (cf. Ramesh and

Dhar 1994). In these ways, DEALSCRIBE supports the asynchronous work of analysts through monitoring and maintaining of development goals.

This section demonstrates how development goal monitoring can play an active role in managing requirements development. Rather than continue with the `Have-UserPriority` example of the previous sections, I introduce a requirements conflict detection and resolution dialog protocol, called Root Requirements Management. This protocol specifies dialog goals that support its particular form of requirements conflict management. Although this protocol is not a standard, it does specify goals concerning analyst interactions (e.g., voting), as well as goal remedy operations. As such, Root Requirements Management demonstrates more fully than other requirements conflict detection and resolution protocols how development goal monitoring can take an active role in managing requirements development.

6.4.1 Root Requirements Management

Root Requirements Management was developed as a dialog protocol for managing requirements conflicts. It is aimed at addressing two basic objectives: conflict understanding and conflict removal. First, Root Requirements Analysis was developed to aid conflict understanding (Robinson and Pawlowski 1998). This technique uncovers requirements conflicts, analyzes them as a group, and directs analysis toward key conflicts. Second, to generate alternative resolutions for each conflict, Conflict-Oriented Requirements Restructuring (CORR) was developed (Robinson and Volkov 1997). To demonstrate how development goal monitoring can apply to more complex protocols, I show in this section how a Root Requirements Management protocol can be defined in terms of goal monitoring. The protocol developed here consists of (1) Root Requirements Analysis, (2) Conflict-Oriented Requirements Restructuring, and (3) resolution selection.

6.4.1.1 Root Requirements Analysis Two objectives of Root Requirements Analysis are (1) to understand the relationships among a project's requirements and (2) to order requirements according to the degree of contention over them. This information can be used to guide other analyses, such as conflict resolution through Conflict-Oriented Requirements Restructuring.

The overall procedure of Root Requirements Analysis is as follows:

1. *Identify root requirements* that cover all requirements in the requirements document. Requirements are generalized to derive these root requirements.

2. *Identify interactions* among root requirements. Root requirements are subjectively compared pairwise to determine the interactions among them; the interaction types are "very conflicting," "conflicting," "neutral," "supporting," and "very supporting."

3. *Analyze the root requirement interactions*. Requirements metrics are derived from the root requirements interactions.

This technique is important in that it provides a systematic method through which requirements conflicts can be surfaced and then systematically selected for efficient resolution.

6.4.1.2 Root Requirements Analysis Metrics Once the root requirements conflicts are identified, they can be used to derive useful metrics. Three that are particularly helpful are relationship count, requirement contention, and average potential conflict. *Relationship count* is simply a count, for each root requirement, of the number of interactions the root requirement has with other root requirements, for each of the five types of interactions. *Requirement contention* is the percentage (expressed as a ratio between zero and one) of all interactions the requirement participates in that are conflicting; thus, if a requirement's contention is one, then it conflicts with every other requirement in the requirements document. Finally, *average potential conflict* is the conflict potential of a requirement averaged across all of its conflicting relationships.

These metrics are useful in determining the order in which conflicts should be resolved. For example, resolving the most contentious requirement first not only directly resolves one conflict but often indirectly resolves others (Robinson and Pawlowski 1998). Thus, resolving high-contention requirements first is a practice of Root Requirements Management that is supported by a dialog goal (`ResolveHighestContentionFirst`, presented in section 6.4.2).

6.4.1.3 Resolution Generation through Requirements Restructuring The purpose of resolution generation is to remove conflict by finding substitute requirements that fulfill the intent of the original requirements, but without their undesired consequences. Resolutions can be generated by altering the structure of the original requirements through transformations. Resolutions fall into three general classes: (1) selection among the original conflicting requirements, (2) selection among restructurings of the conflicting requirements, and (3) selection of run time monitoring and recovery of the original conflicting requirements (Robinson and Volkov 1996). Such resolutions are generated through a combination of two classes of restructuring transformations:

1. *object-restructuring* transformations, which use object relationships (e.g., `isA` and `partOf`) to find related objects for substitution

2. *condition-restructuring* transformations, which distribute conflicting positions across specific contexts, thereby conditionalizing the state under which each requirement will be met

These transformations are applied as a part of the practice of Root Requirements Management. As such, they are supported as remedies of the dialog goal `Resolve-HighestContentionFirst`.

6.4.1.4 Resolution Selection Once resolutions are generated for each conflict, analysts must select which resolution(s) will become part of the next requirements baseline. Voting is an efficient means for group selection. Resolution selection through voting is a subgoal of the Root Requirements Management protocol.

6.4.2 A Dialog Protocol for Root Requirements Management

The following summarizes the dialog practices of Root Requirements Management. These practices can be translated into DealScribe dialog goals as part of the protocol for Root Requirements Management:

1. `RootRequirementsAnalysis`: The analysis portion has three basic subgoals:

a. `DeriveRoots`: No more than 3 percent of all requirements shall lack an associated root requirement.[7]

b. `DeriveInteractions`: No more than 5 percent of all root requirements shall lack an associated description of conflict (called a *requirement interaction*).

c. `DeriveContention`: No more than five new interactions shall exist before requirements contention analysis is conducted.

2. `GenerateResolutions`: Resolutions shall be generated for each requirements conflict according to the following goal:

a. `ResolveHighestContentionFirst`: Resolutions shall be generated for conflicts among the most contentious requirements. To generate resolutions, do the subgoals of generating: object, distribution, and interaction resolutions.

3. `SelectResolutions`: Resolutions shall be selected for each requirements conflict according to the following goal:

a. `VoteOnResolutions`: A vote shall be conducted where there are multiple resolutions of a requirements conflict.

Each of these practices is translated into a DealScribe goal in the following subsections.

6.4.2.1 Root Requirements Dialog Goals The first goal, `DeriveRoots`, simply states that no more than 3 percent of all requirements are to lack an associated root requirement. The definition makes use of `RequirementsWithoutARoot`, which indicates those requirements that have not been analyzed for roots. DealScribe can determine the percentage of requirements (`query2`) that lack an associated root requirement (`query1`) through answers to the following query classes:

```
QueryClass RequirementsWithoutARoot isA Requirement with
  constraint
    noroot : $ not exists r/rootrequirement (r requirements this)
    $
end

QueryClass DeriveRoots in PercentGoal with
  mode
    mode_a : Achieve
  query1
    q1 : RequirementsWithNoRoot
  query2
    q2 : Requirement
  goal_count
    gc : 3
  relation
    c : LessThan
end
```

The goal `DeriveInteractions` is identical to `DeriveRoots` except that `query1` retrieves `RootsWithoutAnInteraction` rather than `RequirementsWithNoRoot`. Also, the `DeriveContention` goal is similar but uses counts instead of percentages. It also has one additional attribute that is not a part of `DeriveRoots`. `DeriveContention`'s `remedy` attribute indicates an operation statement that should be executed automatically if goal failure occurs. Upon goal failure, the dialog goal and results of the goal-checking query are passed to a remedy operation. In this case, upon failure of `DeriveContention`, `ContentionAnalysis` is executed, and the results are asserted to the dialog.

The composite goal for Root Requirements Analysis is the conjunction of the three goals discussed in the previous paragraph, as shown in the following:

```
QueryClass RootRequirementsAnalysis in DialogGoal with
  mode
    mode_a : Achieve
  andGoals
    g1 : DeriveRoots;
    g2 : DeriveInteractions;
    g3 : DeriveContention
end
```

6.4.2.2 Resolution Generation Dialog Goals The resolution generation goal is a bit more complex. Recall that `DeriveContention` contains a degree of inconsistency,

called "contention." The goal ResolveHighestContentionFirst seeks to resolve interactions among the most contentious requirements first. It is defined as follows:

```
Class MostContentiousUnresolvedRequirements isA Requirement with
  constraint
    ConReq : $ (not exists gr1/GenerateResolution (gr1 requirements
    this) and exists thisCon/Integer (this Contention thisCon)) and
    (not exists otherReq/Requirement otherCon/Integer ((otherReq
    Contention otherCon) and (otherCon > thisCon) and not exists
    gr2/GenerateResolution (gr2 requirements otherReq))) $
end

QueryClass ResolveHighestContentionFirst in DialogGoal isA
RequirementInteraction with
  mode
    mode_a : Achieve
  remedy
    r1 : ObjectRestructuring;
    r2 : DistributionRestructuring;
    r3 : InteractionRestructuring
  constraint
    RHF : $ exists req1,req2/MostContentiousUnresolvedRequirements
    ((this requirements r1) and (this requirements r2)) $
end
```

The definition of ResolveHighestContentionFirst makes use of a derived class, MostContentiousUnresolvedRequirements. This class is defined to be those requirements for which there has not been a resolution generated and there does not exist another requirement with a higher level of contention for which there has not been a resolution generated. Once MostContentiousUnresolvedRequirements is defined, specifying the goal ResolveHighestContentionFirst is easy. It is simply those requirements that both (1) are in the MostContentiousUnresolvedRequirements and (2) interact with one another, as denoted by both being in the requirements of the same RequirementInteraction. Thus, ResolveHighestContentionFirst makes use of the dialog forum to specify the goal of always selecting unresolved interactions among requirements with the highest contention.

The dialog goal ResolveHighestContentionFirst specifies three remedies that correspond to the three types of resolution generation methods described in section 6.4.1. When ResolveHighestContentionFirst is monitored and fails, the three operations will be executed.

6.4.2.3 Resolution Selection Dialog Goals The final goal simply specifies that where there is more than one possible resolution for a requirements conflict, stakeholders should vote. The goal `VoteForResolution` specifies this as a counting goal. Whenever the count of potential resolutions for a particular requirements conflict is greater than one, the goal fails, and the remedy `VoteTally` is executed. The operation `VoteTally` simply asserts a statement indicating the number of votes each resolution has received. Of course, voting may be more complex, involving time periods and multiple rounds; however, `VoteForResolution` gives a flavor of how resolution selection can be included in the protocol.

6.4.2.4 Analysis and Resolution Dialog Protocol The final protocol is represented as a composite goal that contains the preceding goals:

```
QueryClass RootRequirementsManagement in DialogGoal with
  mode
    mode_a : Achieve
  andGoals
    g1 : RootRequirementsAnalysis;
    g2 : ResolveHighestContentionFirst;
    g3 : VoteForResolution
end
```

Once so defined, this goal model may be selected as part of the input to run the `GoalCheck` operation. The operation `GoalCheck(RootRequirementsManagement)` can be monitored to ensure updated reports and remedies concerning Root Requirements Management. Of course, `RootRequirementsManagement` is but one goal model. Multiple goal models can coexist simultaneously or at different times. However, it is the responsibility of the user to ensure that multiple goal models do not defeat one another's goals or enter into accidental loops through the interaction of remedy operations.

6.4.2.5 Summary The preceding definition of the Root Requirements Management protocol demonstrates how an interesting dialog protocol can be defined in terms of the dialog metamodel of sections 6.2 and 6.3. The following section demonstrates that such a protocol definition can be effectively monitored as part of a requirements development dialog.

6.4.2.6 Observations on the Use of DealScribe I believe the benefit of dialog goal monitoring increases as confusion about the dialog increases. As the number of requirements, analysts, and their analyses grows, confusion over the current state of requirements can also grow.

Dialog goal monitoring provides assurance that the dialog protocol is being followed. In contrast to operation automation, protocol assurance is directly attributable to dialog goal monitoring. On the other hand, it is difficult to assess what effect this assurance has on analysts. Nevertheless, the following subjective findings may be of interest until subsequent empirical studies can be conducted.

As an analyst, I believe that development goal monitoring provided me with a better understanding of the current development state than I had obtained in a manual case study. Such understanding was gained through warnings that arose in response to goal failures. Once the goal failures were removed, I was assured that the development satisfied the dialog protocol.

I also believe that DealScribe provided me with a better visualization of the requirements development. In figure 6.3, one can see those statements that fail subgoals of a goal tree. Such visibility of goal failure, with its direct linkage to statements, provides tangible assurance that development goals are being tracked. More generally, the collaborative discussion environment of DealScribe provides a visualization of development that is not available in word-processor-based case studies. For example, the presentation of analysis as a hierarchical discussion (see figure 6.6) provides an understanding of how new analysis fits in with older results.

6.4.3 Future Directions

The research on development goal monitoring described in this chapter is continuing along two directions. First, the functionality of the dialog-monitoring system is being extended. Currently, goals are checked and remedied according to a predefined goal tree. Although goal trees can be modified during development, there is no support for doing so. A future version of DealScribe will support dynamic dialog planning. Given predefined remedy operations, with pre- and postconditions, the planner will check goals and dynamically construct and apply operations to reestablish failed goals. This will simplify the expression and execution of dynamic dialog models. Second, a library of predefined development protocols and associated analyses is being constructed. Future work will focus on expanding this library, proving protocol properties, and integrating the application of various requirements techniques under a common dialog protocol.

6.5 Conclusions

A development-goal-monitoring system aimed at supporting multianalyst requirements development has been presented. The conceptual design specifies a multiuser forum of informational and operational statements that can be analyzed and monitored for correspondence with a dialog protocol. The metamodel described and used in the dialog system defines statements whose properties can be formally defined and

automatically checked. Such a metamodel facilitates analysis, as well as system extensibility.

An implementation of the development-goal-monitoring system has been presented in this chapter. DEALSCRIBE demonstrates support of a typology of informational and operational statements, checks and remedies of formal goals, operation execution in response to monitored goals, and hypothetical statement assertion, all in a multiuser Web environment.

An exploratory case study has also been presented. It demonstrates that a goal-monitoring system can provide automation for a multiuser dialog, assurance about compliance with a dialog protocol, and greater understanding of requirements development.

The research presented in this chapter contributes to the field of requirements management by showing how one can formally specify development goals that can be directly monitored as part of a multiuser dialog. The dialog metamodel provides a concise, extensible model for uniformly including informational and operational statements, as well their monitored execution. Development goal monitoring within a collaborative requirements analysis tool can provide a powerful environment for managing development and document inconsistencies.

Notes

1. Defining a dialog protocol as a composite goal makes it possible to define multiple dialog protocols for a forum. Each dialog protocol may be activated (or inactivated) through monitoring operations on its composite dialog goal. Although this feature has been useful in experimenting with dialog protocols, it has not been used in practice.

2. The notation `GoalCheck( HaveUserPriority)` indicates that a specific operation, `GoalCheck`, is executed with the specified arguments, `HaveUserPriority`.

3. The current implementation does provide some support for preventing or halting recursion of an operation. It does not execute a new operation of type `S` if the new operation is activated in response to another statement of type `S` and that other statement was asserted in the last monitored cycle.

4. One could argue that such hypothetical considerations should be made part of the dialog history. An anonymous reviewer of the book remarked that this could be done by managing a history tree. The current implementation uses a linear statement history with hypothetical statements because of the smaller history it entails, the simplicity of the user interface, and the ease of implementation.

5. All components communicate using Portable Operating System Interface (POSIX)-compliant Transmission Control Protocol (TCP) connections.

6. In the ConceptBase query notation, `this` refers to the instance retrieved from the database—in `HaveUserPriority`, a `Requirement`. Other ConceptBase notations include: `x/Type`, which constrains variable `x` to being an instance of class `Type`, and `(this userPriority x)`, which constrains the value `x` of attribute `userPriority` for object `this`.

7. The percentages given here are simply intended to illustrate numeric goals. Percentages in goals determine how much deviation from compliance a dialog may have. Once failure is reached, DEALSCRIBE provides a warning (and possibly a remedy) after each dialog event until there is no goal failure. In a circumstance in which an analyst is required to manually remedy a goal failure, a percentage goal allows an analyst some time prior to the activation of automated warnings. Of course, an analyst can directly ap-

ply `GoalCheck` to a nonpercentage goal (e.g., `RequirementsWithoutARoot`) to immediately determine any failures.

References

Alford, M. 1992. "Strengthening the Systems Engineering Process." *Engineering Management Journal* 4, no. 1: 7–14.

Boehm, B., and H. In. 1996. "Identifying Quality-Requirement Conflicts." *IEEE Software* 13, no. 2: 25–35.

Ceri, S., G. Gottlob, and L. Tanca. 1990. *Logic Programming and Databases*. New York: Springer-Verlag.

Chen, M., and J. Nunamaker. 1991. "The Architecture and Design of a Collaborative Environment for Systems Definition." *Data Base* 22, no. 1/2: 22–29.

Chikofsky, E. J., and B. L. Rubenstein. 1988. "CASE: Reliability Engineering for Information Systems." *IEEE Software* 5, no. 2: 11–16.

Cs3. 2007. "FLEA Overview." Available at: ⟨http://www.cs3-inc.com/flea_overview.html⟩.

Curtis, B., H. Krasner, and N. Iscoe. 1988. "A Field Study of the Software Design Process for Large Systems." *Communications of the ACM* 31, no. 11: 1268–1287.

Davy, C. 1990. "Using CASE to Control a Large Data Analysis Project." In *CASE on Trial*, ed. K. Spurr and P. Layzell, 7–16. New York: Wiley.

Egyed, A., and B. Boehm. 1996. "Analysis of Software Requirements Negotiation Behavior Patterns." Technical report, USC-CSE-96-504, University of Southern California, Los Angeles.

Emmerich, W., A. Finkelstein, C. Montangero, S. Antonelli, S. Armitage, and R. Stevens. 1999. "Managing Standards Compliance." IEEE *Transactions on Software Engineering* 25, no. 6: 836–851.

Fickas, S. and M. S. Feather. 1995. "Requirements Monitoring in Dynamic Environments." In *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE'95)*, 140–147. Los Alamitos, CA: IEEE Computer Society.

Gotel, O., and A. Finkelstein. 1994. "An Analysis of the Requirements Traceability Problem." In *Proceedings of the First IEEE International Conference on Requirements Engineering (RE'94)*, 94–101. Los Alamitos, CA: IEEE Computer Society.

Gotel, O., and A. Finkelstein. 1995. "Contribution Structures." In *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE'95)*, 100–107. Los Alamitos, CA: IEEE Computer Society.

Graf, D. K., and M. M. Misic. 1994. "The Changing Roles of the Systems Analyst." *Information Resources Management Journal* 7, no. 2: 15–23.

Jarke, M., R. Gallersdorfer, M. A. Jeusfeld, M. Staudt, and S. Eherer. 1995. "ConceptBase—A Deductive Object Base for Meta Data Management." *Journal of Intelligent Information Systems* 4, no. 2: 167–192.

Klein, M. 1991. "Supporting Conflict Resolution in Cooperative Design Systems." *Transactions on Systems, Man, and Cybernetics* 21, no. 6: 1379–1390.

Krasner, H., B. Curtis, and N. Iscoe. 1987. "Communication Breakdowns and Boundary Spanning Activities on Large Programming Projects." In *Empirical Studies of Programmers: Second Workshop*, ed. G. Olson, S. Sheppard, and E. Soloway, 47–64. Norwood, NJ: Ablex.

Lempp, P., and L. Rudolf. 1993. "What Productivity Increases to Expect from a CASE Environment: Results of a User Survey." In *Computer Aided Software Engineering (CASE)*, ed. E. J. Chikofsky, 147–153. Los Alamitos, CA: IEEE Computer Society.

Liou, Y. I., and M. Chen. 1993–1994. "Using Group Support Systems and Joint Application Development for Requirements Specification." *Journal of Management Information Systems* 10, no. 3: 25–41.

Lyytinen, K., and R. Hirschheim. 1987. "Information Systems Failures—a Survey and Classification of the Empirical Literature." In *Oxford Surveys in Information Technology* 4, ed. P. Zorkoczy, 257–309. New York: Oxford University Press.

Mazza, C., J. Fairclough, B. Melton, D. De Pablo, A. Scheffer, and R. Stevens. 1994. *Software Engineering Standards*. Upper Saddle River, NJ: Prentice Hall.

Meyer, B. 1986. "On Formalism in Specification." *IEEE Software* 2, no. 1: 6–26.

Mi, P., and W. Scacchi. 1992. "Process Integration in CASE Environments." *IEEE Software* 9, no. 2: 45–53.

Miller, J., D. Palaniswami, A. Sheth, K. Kochut, and H. Singh. 1998. "METEOR$_2$'s Web-Based Workflow Management System." *Journal of Intelligent Information Systems* 10, no. 2: 185–215.

Minker, J. 1996. "Logic and Databases: A 20 Year Retrospective." In *Logic in DataBases*, ed. D. Pedreschi and C. Zaniolo (Lecture Notes In Computer Science 1154), 3–57. New York: Springer.

Mylopoulos, J., A. Borgida, M. Jarke, and M. Koubarakis. 1990. "Telos: Representing Knowledge About Information Systems." *ACM Transactions on Information Systems* 8, no. 4: 325–362.

Norman, R. J., and J. F. Nunamaker, Jr. 1989. "CASE Productivity Perceptions of Software Engineering Professionals." *Communications of the ACM* 32, no. 9: 1102–1108.

Osterweil, L., and S. Sutton 1996. "Using Software Technology to Define Workflow Processes." In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-art and Future Directions*, ed. A. Sheth, 157–160.

Potts, C., K. Takahashi, and A. I. Antón. 1994. "Inquiry-Based Requirements Analysis." *IEEE Software* 11, no. 2: 21–32.

Ramesh, B., and V. Dhar. 1994. "Representing and Maintaining Process Knowledge for Large-Scale Systems Development." *IEEE Expert* 9, no. 2: 54–59.

Robey, D., D. L. Farrow, and C. R. Franz. 1989. "Group Process and Conflict During System Development." *Management Science* 35, no. 10: 1172–1191.

Robinson, W. N., and S. Pawlowski. 1998. "Surfacing Root Requirements Interactions from Inquiry Cycle Requirements." In Proceedings *of the Third International Conference on Requirements Engineering (ICRE'89)*, 82–89. Los Alamitos, CA: IEEE Computer Society.

Robinson, W. N., and S. D. Pawlowski. 1999. "Managing Requirements Inconsistency with Development Goal Monitors." *IEEE Transactions on Software Engineering* 25, no. 6: 816–835.

Robinson, W. N., and S. Volkov. 1996. "Conflict-Oriented Requirements Restructuring." GSU CIS Working Paper 96-15, Georgia State University, Atlanta, GA.

Robinson, W. N., and S. Volkov. 1997. "A Meta-Model for Restructuring Stakeholder Requirements." *Proceedings of the Nineteenth International Conference on Software Engineering*, 140–149. Los Alamitos, CA: IEEE Computer Society.

Robinson, W. N., and S. Volkov. 1998. "Supporting the Negotiation Life Cycle." *Communications of the ACM* 41, no. 5: 95–102.

Sheth, A., D. Georgakopoulos, S. Joosten, M. Rusinkiewicz, W. Scacchi, J. Wileden, and A. Wolf. 1996. "Report from the NSF Workshop on Workflow and Process Automation in Information Systems." *ACM SIGMOD Record* 24, no. 4: 55–67.

Telelogic. 2007. "Increase Quality with Requirements Management and Traceability." Available at ⟨http://www.telelogic.com/Products/doors/index.cfm⟩.

Vessey, I., and A. P. Sravanapudi. 1995. "CASE Tools as Collaborative Support Technologies." *Communications of the ACM* 38, no. 1: 83–95.

Walz, D. B., J. J. Elam, H. Krasner, and B. Curtis. 1987. "A Methodology for Studying Software Design Teams: an Investigation of Conflict Behaviors in the Requirements Definition Phase." In *Empirical Studies of Programmers: Second Workshop*, ed. G. Olson, S. Sheppard, and E. Soloway, 83–89. Norwood, NJ: Ablex.

Walz, D. B., J. J. Elam, and B. Curtis. 1993. "Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration." *Communications of the ACM* 36, no. 10: 63–77.

7 Definition of Semantic Abstraction Principles

Mohamed Dahchour and Alain Pirotte

This chapter presents a case study of the method-engineering approach to the design and implementation of new abstraction principles. Materialization is presented as an abstraction pattern relating a class of categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars) that cannot be wholly expressed in terms of other generic patterns, like classification and specialization. The structural properties of materialization are expressed by attributes and its behavioral properties by deductive rules and integrity constraints. The case study illustrates a possible extension of the semantic expressive power of various modeling frameworks and CASE environments.

7.1 Introduction

Conceptual modeling is the activity of formalizing some aspects of physical and social systems for purposes of understanding and communication. Conceptual models are typically built in the early stages of system development, preceding design and implementation. But conceptual models can also be useful even if no system is contemplated: They then serve to clarify ideas about structure and functions in a perception of a part of the world.

Advances in conceptual modeling involve narrowing the gap between real-world concepts and their representation in conceptual models by identifying powerful abstraction mechanisms allowing an accurate and intuitive representation of application domains (Dahchour 2001; Dahchour, Pirotte, and Zimányi 2005; Mylopoulos 1998). Thus, more powerful conceptual models help increase mastery of the software-development process and the quality of the final applications.

Generic relationships in object and semantic models are such powerful abstraction mechanisms. They are high-level templates for relating classes of objects. Well-known generic relationships include generalization, classification, and aggregation. Recent research on conceptual modeling has studied other generic relationships like materialization (Dahchour, Pirotte, and Zimányi 2002a; Pirotte et al. 1994),

ownership (Yang et al. 1994), role (Dahchour, Pirotte, and Zimányi 2002b; Wieringa, De Jonge, and Spruit 1995), grouping (Motschnig-Pitrik and Storey 1995), viewpoint (Bertino 1992; Motschnig-Pitrik and Mylopoulos 1996), and versioning (Andonoff et al. 1996). These generic relationships naturally model phenomena typical of complex application domains whose semantics escape direct representation with classical relationships. A review of generic relationships can be found in Dahchour 2001 and Dahchour, Pirotte, and Zimányi 2005.

Languages for conceptual modeling can substantially ease the task of modelers if they are enriched with a variety of generic relationships. This chapter deals with one such extension, called materialization. Materialization is a powerful and ubiquitous semantic pattern that relates a class of abstract categories (e.g., models of cars) and a class of more concrete objects (e.g., individual cars). Its semantics are defined in terms of the usual `isA` (generalization) and `is-of` (classification) abstractions and a class/metaclass correspondence. New and powerful attribute propagation (i.e., inheritance) mechanisms are naturally associated with materialization.

Like other classical abstractions, materialization is a generic relationship, that is, a template to be instantiated in applications. Application classes can thus be provided with structure and behavior consistent with the semantics of materialization. As is usually done with generic relationships, the same name (namely, materialization) is used for both the generic relationship and its concrete realizations in applications.

This chapter presents materialization and shows how its generic semantics can be integrated into ConceptBase. The structural semantics of materialization are represented by a collection of meta-attributes and its behavioral semantics by a set of constraints and deductive/active rules. Those generic semantics defined at the metalevel are then automatically enforced at the class level.

A formalization of materialization along the lines of Telos, without taking into account implementation issues, is presented in Dahchour 1998. A metaclass implementation of materialization into VODAK, an object database system representing the behavior of objects with messages and methods, is reported in Dahchour 2001 and Dahchour, Pirotte, and Zimányi 2002a. The metaclass approach has also been used to implement some other generic relationships (e.g., Dahchour, Pirotte, and Zimányi 2004; Halper, Geller, and Perl 1998; Klas and Schrefl 1995; Gottlob, Schrefl, and Röck 1996).

7.2 The Need for Materialization

Consider the problem of keeping track of cars in a dealer's inventory. A simple solution is provided by the *instantiation* mechanism illustrated in figure 7.1. The application designer builds class `Car`, and all individual cars are created as instances of `Car`. Class `Car` represents information like model name, sticker price, gas mileage, manu-

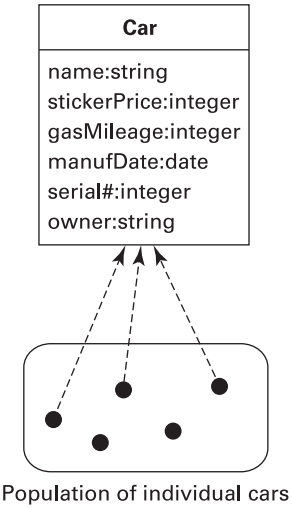


Figure 7.1
The instantiation solution

facturing date, serial number, and owner identification. The main advantage of such a representation is that it makes it easy to add a new car to the inventory. A severe disadvantage, however, is that some information is repeated for cars of the same model. For example, all cars of model `FiatRetro` have the same model name, sticker price, and gas mileage.

An alternative solution based on simple *inheritance* is shown in figure 7.2. Unlike in the first solution, in the simple-inheritance solution, the application designer builds class `Car` as a superclass with one subclass for each car model, such as the subclasses `FiatRetro_Cars` and `Wild2CV_Cars`. Class `Car` represents general information about individual cars, namely, manufacturing date, serial number, and owner identification. In addition to the properties inherited from `Car`, its subclasses define as class attributes common properties (model name, sticker price, gas mileage) of all cars of the same model.

The main advantage of this solution is that it no longer exhibits the redundancy present in the first solution: The common properties of all cars of the same model are not duplicated for each individual car but are instead stored once and for all as class attributes in the corresponding subclass. The solution introduces a new kind of redundancy, however, in that the subclasses are structurally equivalent to one another.

This leads to a third solution, sketched in figure 7.3, which adds to the structure of the second solution (i.e., a superclass `Car` and a subclass of `Car` for each car model)

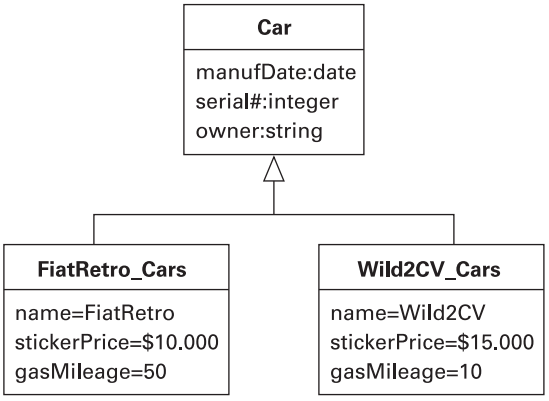


Figure 7.2
The simple-inheritance solution

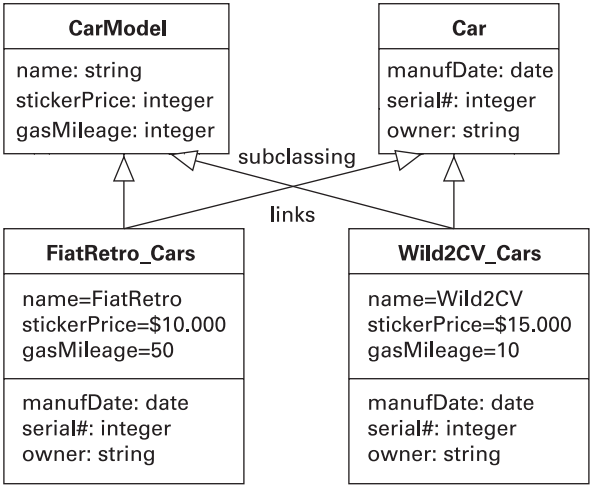


Figure 7.3
The multiple-inheritance solution

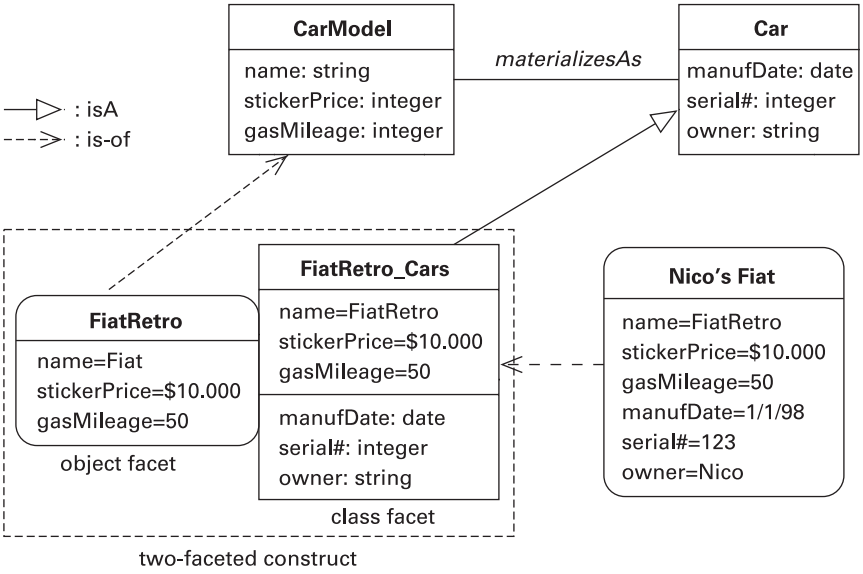


Figure 7.4
Materialization between `CarModel` and `Car`

another superclass, namely, `CarModel`. Class `CarModel` gathers all information that characterizes not individual cars, but collections of car models. Each specific class representing a collection of cars of the same model (e.g., `FiatRetro_Cars`) is a subclass of both `Car` and `CarModel` (with multiple inheritance). The presence of class `CarModel` eliminates the redundancy of the second solution.

Still, this third solution is not without problems. First, the subclasses of `Car` look like instances of `CarModel`. More precisely, class `FiatRetro_Cars`, for example, can be viewed both as an instance of `CarModel` and as a subclass of `Car`. Thus, an instantiation link between `FiatRetro_Cars` and `CarModel` seems more appropriate than a subclassing one. Second, `CarModel` and `Car` are not at the same level of abstraction, as `CarModel` is *more* abstract than `Car`. Third, intuitively, the semantic relationship between `CarModel` and `Car` looks as though it can be abstracted into a generic relationship, but classical relationships cannot capture it. The relationship between `CarModel` and `Car` is called *materialization* (see figure 7.4). It is thus a special relationship between two classes in which one (here, `CarModel`) is more abstract than the other (here, `Car`).

Each subclass of figure 7.3 becomes a two-faceted construct as shown in figure 7.4 with an *object facet* (e.g., `FiatRetro`) that is an instance of `CarModel` and a *class facet* (e.g., `FiatRetro_Cars`) that is a subclass of `Car`. (Of the two subclasses in figure 7.3, only the representation of `FiatRetro_Cars` is shown in figure 7.4.)

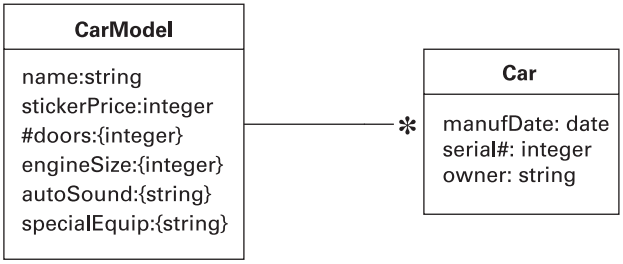


Figure 7.5
An example of materialization

7.3 Intuitive Definition

The previous section has argued for the need for materialization to represent semantics that cannot be captured by classical relationships. This section presents an intuitive definition for materialization, and its formal semantics are analyzed in section 7.4.

Intuitively, materialization relates a class of categories to a class of more concrete objects analyzed with those categories. Figure 7.5 shows a materialization linking classes `CarModel` and `Car`. `CarModel` is the more abstract[1] class and `Car` is the more concrete class of materialization. A materialization link is drawn as a straight line with an asterisk on the side of the more concrete class.

`CarModel` represents information typically displayed in the catalog of car dealers, namely, name and price of a car model, and lists of options for number of doors, engine size, sound equipment, and special equipment. Class `Car` represents information about individual cars, namely, manufacturing date, serial number, and owner identification.

Figure 7.6 shows an instance `FiatRetro` of `CarModel` and an instance `Nico's Fiat` of `Car`, of model `FiatRetro`. Intuitively, the materialization `CarModel—*Car` expresses that every concrete car (e.g., `Nico's Fiat`) has exactly one model (e.g., `FiatRetro`), whereas there can be any number of cars of a given model.

A further intuition about abstractness/concreteness is that each car is a concrete realization (or materialization) of a given car model, of which it "inherits" a number of properties in several ways:

- `Nico's Fiat` directly inherits the `name` and `stickerPrice` of its model `FiatRetro`; this mechanism is called `Type1` attribute propagation (or `T1` propagation for short).
- `Nico's Fiat` has attributes `#doors`, `engineSize`, and `autoSound` whose values are selections among the options offered by multivalued attributes with the same

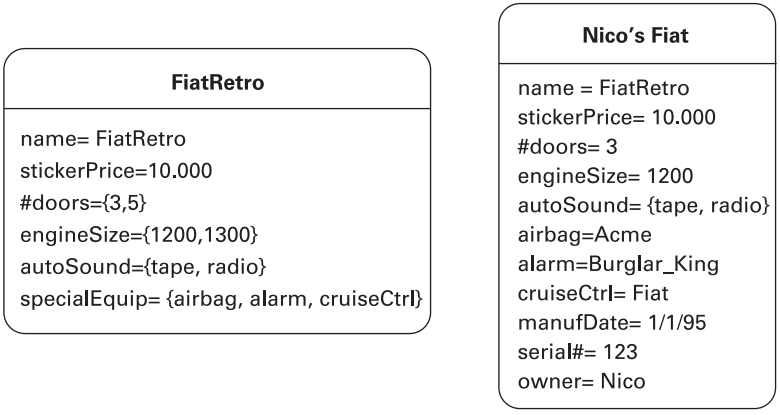


Figure 7.6
Instances of `CarModel` and `Car` from figure 7.5

name in `FiatRetro`; this is called `Type2` (or `T2`) attribute propagation. For example, the value `{1200,1300}` of `engineSize` for `FiatRetro` indicates that each `FiatRetro` car comes with either `engineSize=1200` or `engineSize=1300` (e.g., `1200` for `Nico's Fiat`). Thus, the set value `{1200,1300}` of `engineSize` for `FiatRetro` serves as domain, or type, for the `engineSize` attribute of a subclass of class `Car` consisting of cars with model `FiatRetro`.

• The set value `{airbag, alarm, cruiseCtrl}` of attribute `specialEquip` for `FiatRetro` means that each car of model `FiatRetro` comes with three pieces of special equipment: an airbag, an alarm system, and a cruise control system. Thus, `Nico's Fiat` has three new attributes named `airbag`, `alarm`, and `cruiseCtrl`, whose suppliers are, respectively, `Acme`, `Burglar_King`, and `Fiat`. Other `FiatRetro` cars may have different suppliers for their special equipment, and cars of models other than `FiatRetro` may have a different set of pieces of special equipment. This mechanism is called `Type3` (or `T3`) attribute propagation.

In addition to attributes propagated from `FiatRetro`, `Nico's Fiat` of course has a value for attributes `manufDate`, `serial#`, and `owner` of `Car`. The semantics of attribute propagation are defined in section 7.4.2.

Materializations can be involved in compositions, in which the concrete class of one materialization is also the abstract class of another one, and so on. Figure 7.7 shows a composition of two materializations. It deals with theatre `Plays` with a `title`, an `author`, and a set of main `roles`. `Plays` materialize as `Settings` that add production decisions for a theatrical `season`: a `troupe`, a `director`, and a set of actors for each `role`. `Settings` in turn materialize as `Performances`, with a `theatre` in which the performance takes place, a calendar `date`, the attendance

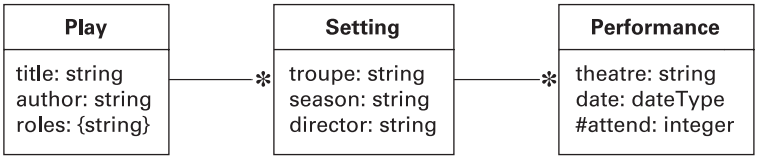


Figure 7.7
Composition of materializations

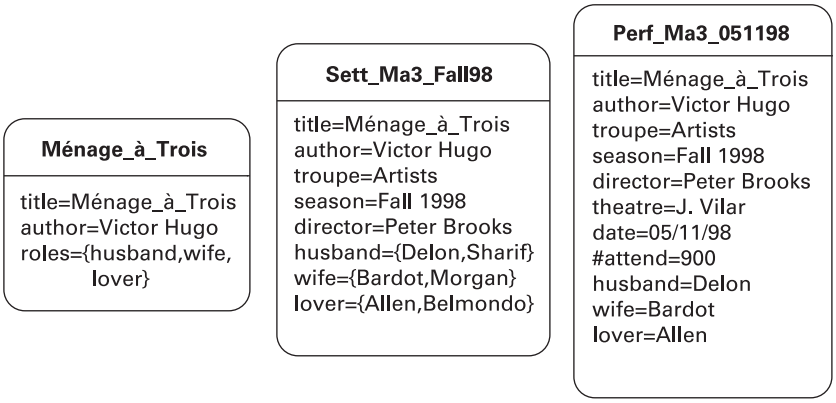


Figure 7.8
Instances of `Play`, `Setting`, and `Performance` from figure 7.7

`#attend` on that date, and each role of `Play` assigned to a specific actor for each `Performance`. A class without a more abstract class (like `Play` in figure 7.7) is called the root of the hierarchy, and a class without a more concrete class (like `Performance`) is a leaf.

Figure 7.8 shows an instance `Ménage_à_Trois` of `Play`, an instance `Sett_Ma3_Fall98` of `Setting`, associated with `Ménage_à_Trois`, and an instance `Perf_Ma3_051198` of `Performance`, associated with `Sett_Ma3_Fall98`. `Ménage_à_Trois` is an ordinary instance of `Play` (see figure 7.7). `Sett_Ma3_Fall98` similarly holds values for the attributes of `Setting`; in addition, it inherits the value of attributes `title` and `author` of `Ménage_à_Trois`; it also creates three new attributes (`husband`, `wife`, and `lover`) from the value of `roles` in `Ménage_à_Trois`, and it assigns them a domain (`{Delon, Sharif}`, `{Bardot, Morgan}`, and `{Allen, Belmondo}`, respectively) for their instances. `Perf_Ma3_051198` holds values for the attributes of `Performance` (see figure 7.7); it inherits from `Sett_Ma3_Fall98` the values of attributes `title`, `author`, `troupe`, `season`, and `director`; and it instantiates attributes `husband`, `wife`, and `lover` of `Sett_Ma3_Fall98`.

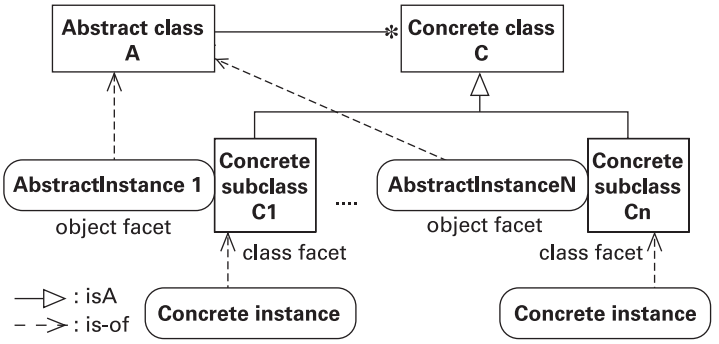


Figure 7.9
Semantics of materialization

Abstract classes can materialize into several concrete classes. For example, data for a movie rental store could involve a class `Movie`, with attributes `director`, `producer`, and `year`, that independently materializes into classes `VideoTape` and `VideoDisc` (i.e., `VideoTape*`—`Movie`—`*VideoDisc`). `VideoTapes` and `VideoDiscs` could have attributes like `inventory#`, `system` (e.g., `PAL`, `NTSC` for `VideoTape`), `language`, `availability` (i.e., in store or rented), and so on.

7.4 Precise Semantics

We now summarize the necessary elements for a formal definition of materialization. Materialization is a binary relationship between two classes `A` and `C`, where `A` is more abstract than `C` (or `C` is more concrete than `A`). Abstractness/concreteness is a user-specified partial order consistent with the cardinalities and the attribute propagation mechanisms of materialization. The materialization relationship has cardinality $(1,1)$ on the side of the concrete class `C` and cardinality $(0,n)$ on the side of the abstract class `A`.

7.4.1 Two-Faceted Constructs

The semantics of materialization are conveniently defined as a combination of the usual `isA` (generalization) and `is-of` (classification) generic relationships and a class/metaclass correspondence, as shown in figure 7.9. In a system with metaclasses, a class can also be seen as an object. *Two-faceted constructs* make that double role explicit. Each two-faceted construct is a composite structure comprising an object (the object facet) and an associated class (the class facet). To underline its double role, we draw a two-faceted construct as an object box adjacent to a class box.

The semantics of materialization A—*C are expressed with a collection of two-faceted constructs as follows. Each object facet is an instance of abstract class A, and the associated class facet is a subclass of concrete class C. Materialization induces a partition of C into a family of subclasses $\{C_i\}$ such that each C_i is associated with exactly one instance of A. Subclasses C_i inherit attributes from C through the classical inheritance mechanism of the isA link. They also "inherit" attributes from A, through the mechanisms of attribute propagation described in the next section. Objects of C, with attribute values "inherited" from an instance of A, are ordinary instances of the class facet associated with that instance of A.

Of course, only application classes, such as A and C (e.g., CarModel and Car), appear in conceptual schemas. The two-faceted construct machinery is managed by the implementation described later and is invisible to users. For them, attribute propagation is built in, and instances of application classes, such as Nico's Fiat in figure 7.6, come with attribute values propagated from their abstract instances through materialization links.

Figure 7.10 sketches the semantics of the materialization of figure 7.5. FiatRetro, an instance of CarModel, is the object facet of a two-faceted construct whose class facet is FiatRetro_Cars, a subclass of Car, describing all instances of Car with model FiatRetro. Wild2CV is another instance of CarModel, and Guy's 2CV is an instance of class facet Wild2CV_Cars. For users, Nico's Fiat and Guy's 2CV are instances of Car, with an instantiation mechanism that integrates attribute propagation, just as instantiation in object models with generalization integrates inheritance from a superclass to its subclasses. In our semantics and their implementation, Nico's Fiat and Guy's 2CV are instances of FiatRetro_Cars and Wild2CV_Cars, respectively.

Similarly, figure 7.11 illustrates the semantics of a composition of two materializations, by displaying one two-faceted construct for each materialization. Ménage_à_

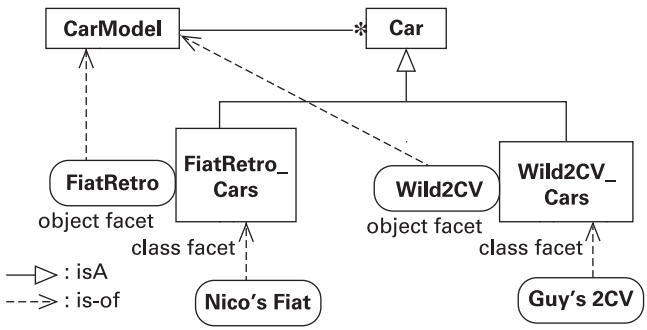


Figure 7.10
Semantics of materialization from figure 7.5

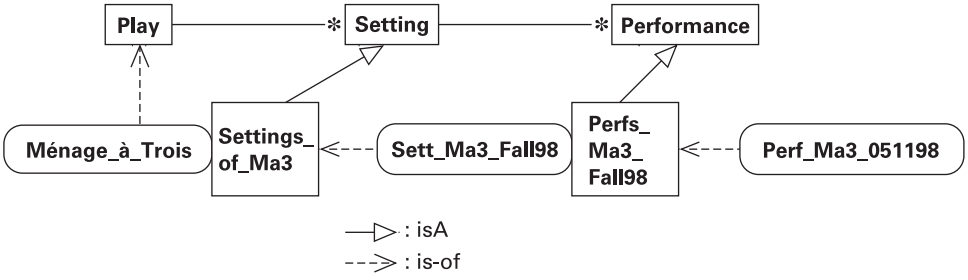


Figure 7.11
Semantics of materialization from figure 7.7

`Trois` is an instance of `Play`. For users, `Sett_Ma3_Fall98` and `Perf_Ma3_051198` are instances of `Setting` and `Performance`, respectively. Our semantics describe them as instances of class facets `Settings_of_Ma3` and `Perfs_Ma3_Fall98`, respectively.

7.4.2 Attribute Propagation

Objects of a concrete class naturally "inherit" information from objects of the corresponding abstract class, as illustrated in section 7.3. We use, from now on, the term "attribute propagation" for the mechanisms associated with materialization and reserve "inheritance" for the usual mechanism for propagating attributes and methods from a superclass to its subclasses in a generalization.

Attribute propagation with materialization is precisely defined as a transfer of information from an abstract object to its associated class facet in a two-faceted construct, as illustrated in figures 7.12 and 7.13. (For clarity, attribute propagation types are shown on abstract classes, although this information really belongs to the materialization links.)

The following definitions regarding attributes will be useful. A *class attribute* of a class `C` has the same value for all instances of `C`; an *instance attribute* of `C` has its value defined for each instance of `C`. Attributes can be *monovalued* (i.e., their value is a single atomic value) or *multivalued* (i.e., their value is a set, possibly empty or singleton, of atomic values).

7.4.2.1 T1 Propagation For users, the T1 propagation mechanism characterizes the plain transfer of an attribute value from an instance of the abstract class to associated instances of the concrete class. In our semantics, the value of a (monovalued or multivalued) attribute is propagated from an object facet to its associated class facet as a class attribute (i.e., its value is the same for all instances of the class facet). For example, the values of the monovalued attributes `name` and `stickerPrice`

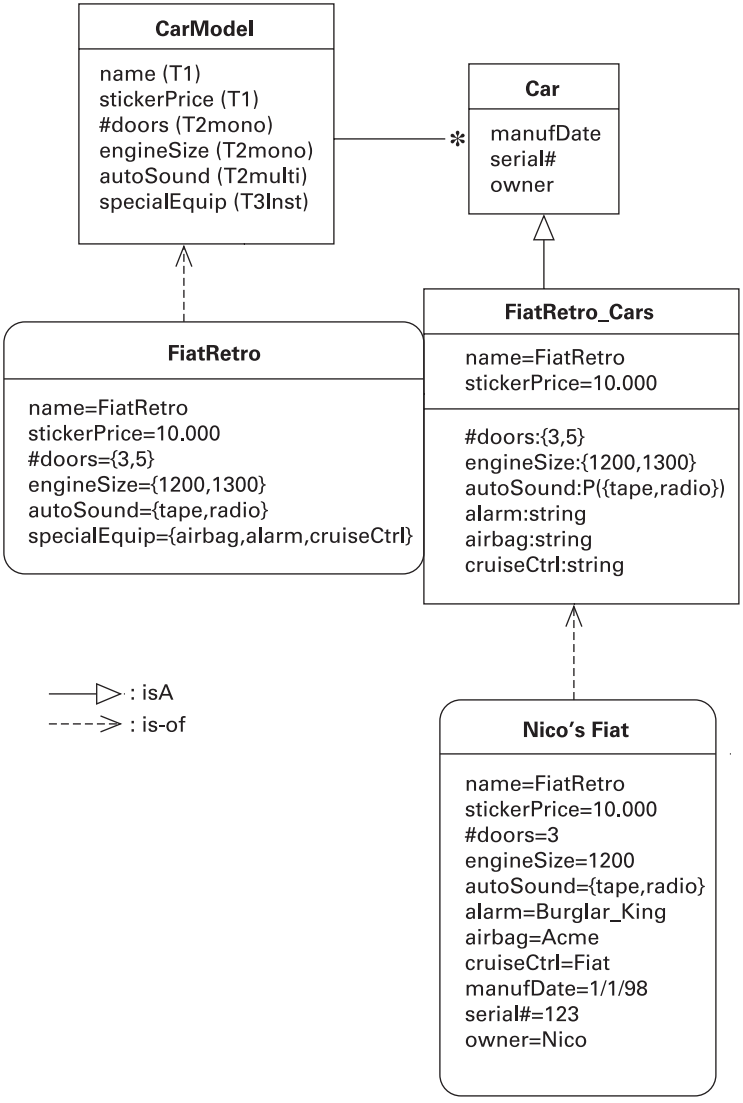


Figure 7.12
Attribute propagation between `CarModel` and `Car`

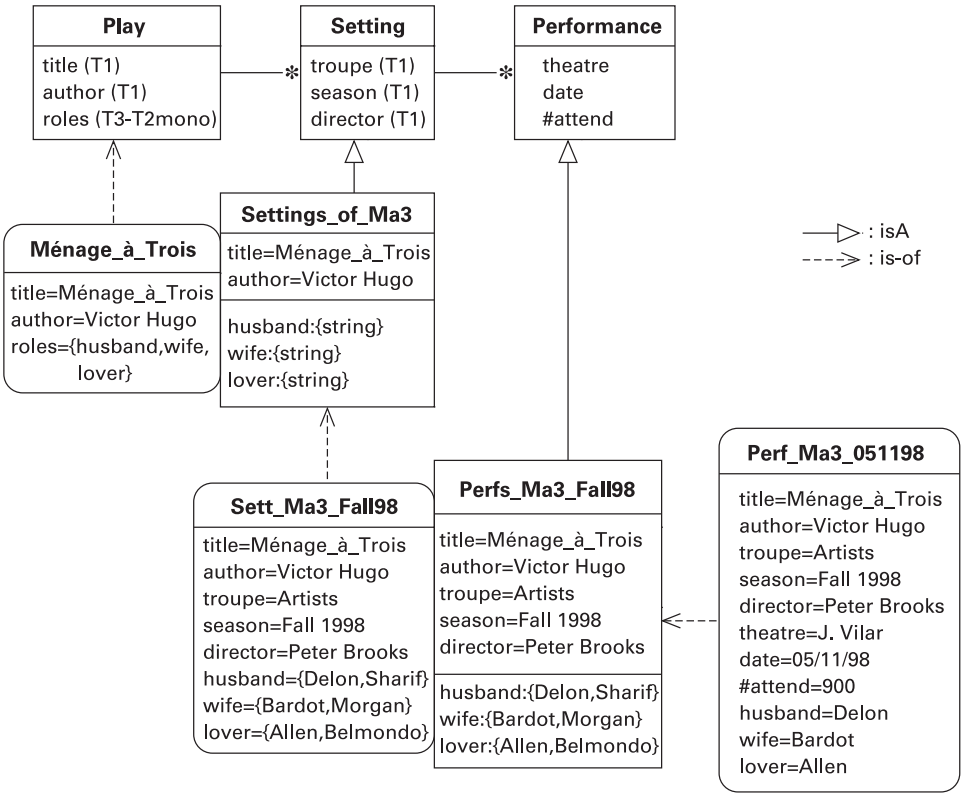


Figure 7.13
Attribute propagation in the composition `Play—*Setting—*Performance`

(`FiatRetro` and `10.000`, respectively) in object facet `FiatRetro` propagate as values of class attributes with the same name in class facet `FiatRetro_Cars` (see figure 7.12). The mechanism is identical for multivalued attributes.

7.4.2.2 T2 Propagation The T2 propagation mechanism concerns multivalued attributes of the abstract class `A`. For users, their value for an instance of `A` determines the domain (or type) of instance attributes with the same name, monovalued or multivalued, in the concrete class `C`. The associated propagation types will be named `T2mono` and `T2multi`, respectively. Again, our semantics go through abstract objects and associated class facets.

An example of `T2mono` propagation is exhibited by `engineSize`, a multivalued attribute of `CarModel` (see figure 7.12). Its set value, noted `engineSize={1200, 1300}`, for the `FiatRetro` object facet is the domain of values for a monovalued

instance attribute with the same name in the associated class facet FiatRetro_Cars, where this is noted engineSize:{1200,1300}. Thus, each FiatRetro car comes either with engineSize=1200 or with engineSize=1300.

An example of T2multi propagation is exhibited by autoSound, a multivalued attribute of CarModel. The set value {tape,radio} of autoSound in object facet FiatRetro indicates that each FiatRetro car comes with either tape, or radio, or both, or nothing at all as autoSound. The associated class facet FiatRetro_Cars has a multivalued instance attribute autoSound with the powerset P({tape, radio}) as its domain.

7.4.2.3 T3 Propagation The T3 propagation mechanism is more elaborate. It also concerns multivalued attributes of the abstract class, whose value is always a set of strings. Each element in the set value of an attribute for an object facet generates a new instance attribute in the associated class facet. The domain of generated attributes must be specified in the definition of the materialization.[2]

For example, attribute specialEquip of CarModel propagates with T3 to Car (see figure 7.12). Its set value {airbag,alarm,cruiseCtrl} for object facet FiatRetro generates three new monovalued instance attributes of domain string, named airbag, alarm, and cruiseCtrl, in the associated class facet FiatRetro_Cars. This propagation type will be called T3Inst; it is the only possible T3 propagation for simple materializations.

7.4.2.4 Further Propagation For a composition A—*C—*D of two materializations, attributes propagated from A to C via A—*C further propagate to D via C—*D. Attributes that propagate from A with T1 are class attributes in C and thus also in D. Attributes that propagate from A with T2 or T3 produce instance attributes in C. If the latter are monovalued, they propagate with T1 to D. If they are multivalued, then they can propagate with T1, T2, or T3 to D.

For example, attribute roles of Play propagates with T3 in Play—*Setting and generates multivalued attributes of domain {string} in Setting. These attributes propagate with type T2mono in Setting—*Performance, producing monovalued instance attributes in Performance (see figure 7.13).

Thus, by T3 propagation, three new multivalued attributes are generated in class facet Settings_of_Ma3 from the set value {husband,wife,lover} of attribute roles in the Ménage_à_Trois instance of Play. Their value in an instance Sett_Ma3_Fall98 of Settings_of_Ma3 is a set of names of actors available for playing each of the husband, wife, and lover roles during a specific theater season (namely, Fall 1998). Then, by T2 propagation, class facet Perfs_Ma3_Fall98 has three monovalued instance attributes, named husband, wife, and lover, whose do-

main is the value of the corresponding attribute in `Sett_Ma3_Fall98`. Finally, one among the actors available for each role is chosen for each performance (e.g., `Delon` as `husband` on `05/11/98` as shown in `Perf_Ma3_051198`). This propagation type of `roles` from `Play` to `Setting` and to `Performance` will be denoted `T3-T2mono`, with the hyphen signaling two-level propagation. Other two-level propagation types work as expected.

7.5 More Examples of Materialization

Materialization provides an extra degree of freedom for building conceptual schemas directly reflecting concepts that are natural in application domains. In summary, for class `C` to materialize class `A` (`A`—*`C`), `C` must be more concrete than `A` in the partial order expressing abstractness. Materialization induces a partition of `C` into a family of subclasses, each associated with exactly one instance of `A`. Cardinalities must be $(1,1)$ on the side of `C` and $(0,n)$ on the side of `A`.

Instances of materialization are ubiquitous, as illustrated by the following examples:

- Modeling air travel can involve a concept of itinerary (from an origin to a destination, with a distance, etc.), materialized as a class of flights (for an airline, with a price, on certain days of the week, periods of the year, etc.), itself materialized as a class of flights for specific calendar days (with a date, an aircraft, a crew, etc.).
- News items can materialize as articles in a particular edition of a newspaper, in turn materialized as physical copies of the newspaper.
- Stories can materialize as book titles (e.g., in publisher catalogs), which materialize as book copies (e.g., in library inventories). Stories can also materialize as theater plays, themselves materialized as performances, a variant of the example presented earlier (see figure 7.7). Movies are another materialization of stories; they can in turn materialize as video titles, which can materialize as physical tapes and discs available in a video rental store. Books in a library or in a bookstore can also be classified according to literary genre (e.g., drama, reference, travel). In a library, they can differ according to their borrowing status (e.g., duration, price, reader privileges).
- For our running example with car models and cars, class `CarModel` can materialize as both brochures and videos presenting the models.
- Film negatives can materialize as positive prints, differing in size, shades of colors, etc.
- Source for text formatters (e.g., LaTeX, HTML) materializes into printed versions of documents of various sizes and shapes.

- Forms (e.g., income tax forms) materialize as filled-in forms (e.g., income tax returns).
- Constitutions materialize into laws, in turn materialized as operational regulations.

Thus, materialization expresses various nuances of meta-information. The most common relationship is *classification*, between categories and concrete objects classified into those categories. Materialization also frequently characterizes *embodiment*, the relationship between classes of objects and their common abstract definition, with the possibility of the relationship's being associated with a *transformation* of objects to produce more detailed objects. Pirotte et al. 1994, also introduces the materialization of relationships (e.g., aggregation) and the materialization of constraints.

7.6 Related Work

An interest in capturing the semantics of materialization has been intuitively perceived, with different names, for as many as twenty years, according to Tabourier (1997). Unlike our own work, little research has addressed the definition of a generic relationship for extending conceptual modeling languages with those semantics. Also, the works referenced in this section view abstractness and concreteness as essentially absolute properties, unlike our own work, which treats abstractness/concreteness as a partial order.

Two constructs related to materialization are mentioned in Object Modeling Technique (OMT) (Rumbaugh et al. 1991) and referred to as *metadata* and *homomorphisms*. The term "materialization" was introduced and informally characterized in Goldstein and Storey 1994. The nearly formal presentation in Pirotte et al. 1994 subsumed that definition. Since then, we have been refining the semantics of materialization and have implemented it in various settings (Dahchour 2001; Dahchour, Pirotte, and Zimányi 2002a; see also Kolp 1999; Zimányi 1997). We describe our implementation of materialization in ConceptBase in section 7.7. The *power types* of Martin and Odell 1995 (252) also catch the basic idea: "a power type is an object type whose instances are subtypes of another object type." A power type is, in our terms, the abstract class of a materialization, and the other object type is its concrete class. We find it more appropriate to attach the semantics to the relationship than to the abstract class. Also, our two-faceted constructs clearly show that even if they are tightly related, the instances of the power type (the object facets) are not the same as the subtypes of the concrete class (the class facets). Indeed, our semantics implement attribute propagation, not discussed in Martin and Odell 1995, as taking place between an object facet and its associated class facet. Semantics similar to those of power types are described as a "type object" design pattern in Johnson and Woolf 1998. A

knowledge level for object types and an operational level for objects are distinguished in Fowler 1997, in the spirit of materialization. Several examples of what we call materialization are presented in Hay 1996. The two directed mappings equivalent to materialization are referred to as "is an example of" and "is embodied in." Similar semantics are analyzed as the "intension" and "extension" of concepts in Al-Jadir et al. 1995 and Falquet, Léonard, and Sindayamaze 1994.

Several patterns frequently occurring in the real world are described in Coad, North, and Mayfield 1995; the closest to materialization is called *item-description* pattern. A library of generic relationships for analysis is suggested in Kilov and Ross 1994; the *reference* association is closest to materialization, but it is defined somewhat informally and without attribute propagation.

7.7 Implementation of Materialization

This section shows how the semantics of materialization can be integrated into ConceptBase.

sources

7.7.1 Class-Level Semantics

The generic semantics of materialization are captured by meta-attribute `materializes` relating two metaclasses `AClass` and `CClass` representing, respectively, the roles of abstract and concrete classes for materializations:

```
AClass in Class with
  attribute
    materializes: CClass
end

CClass in Class end
```

7.7.1.1 Definition of the Materialization Characteristics The following materialization characteristics are defined as attributes of meta-attribute `materializes`:

```
AClass!materializes with
  attribute
    inhAttrT1: Attribute1Def;
    inhAttrT2: Attribute2Def;
    inhAttrT3: Attribute3Def
end

Attribute1Def in Class with
  attribute
```

```
    attrProp: Attribute
end

Attribute2Def in Class with
  attribute
    attrProp: Attribute;
    derivAttr:String {* mono, multi *}
end

Attribute3Def in Class with
  attribute
    attrProp: Attribute;
    genAttrType: TypeDef;
    genAttrPropag: String
    {* T3Inst, T3-T2mono, T3-T2multi *}
end

TypeDef in Class end
```

Meta-attribute `materializes` (accessed as `AClass!materializes` in `AClass`) is declared as a metaclass to carry the semantics of all materialization relationships. Metaclass `AClass!materializes` defines specific attributes labeled `inhAttrT1`, `inhAttrT2`, and `inhAttrT3` specifying the modes for propagating attributes of the abstract class to the corresponding concrete class. We show later that there is no need to define a specific attribute to represent materialization cardinalities. In fact, as the cardinality at the concrete-class side is always $(1,1)$ and that at the abstract-class side is often $(0,n)$, it is more suitable to represent that information by means of constraints attached to metaclass `CClass`.

The domains (or *destinations* in the terminology of ConceptBase) of attributes `inhAttrT1`, `inhAttrT2`, and `inhAttrT3` are defined as follows:

- `Attribute1Def` specifies attributes propagating with `T1`.
- `Attribute2Def` specifies attributes propagating with `T2` and the kind `derived-Attr` (monovalued or multivalued) of the derived instance attribute.
- `Attribute3Def` specifies attributes propagating with `T3`, the value type `gen-AttrType` (`TypeDef`), and a propagation type `genAttrPropag` for the generated attributes. The possible values for `genAttrPropag` are `T3Inst`, `T3-T2mono`, and `T3-T2multi` (see section 7.4.2).

Note that meta-attribute `inhAttrT1` (respectively, `inhAttrT2` and `inhAttrT3`) can be instantiated in applications with as many `T1` (respectively, `T2` and `T3`) attributes as needed. The following shows a partial implementation of the example `CarModel`—`*Car`:

```
CarModel in AClass with
  materializes
    materializesCar: Car
end

Car in CClass end

CarModel!materializesCar with
  inhAttrT1
    T11:T1Name;
    T12:T1StickerPrice
  inhAttrT2
    T21:T2Doors;
    T22:T2EngineSize;
    T23:T2AutoSound
  inhAttrT3
    T31:T3SpecialEquip
end

T1Name in Attribute1Def with
  attrProp
    attrName:CarModel!name
end
{* T1StickerPrice: Idem *}

T2Doors in Attribute2Def with
  attrProp attrDoors:CarModel!#doors
  derivAttr derivDoors:"mono"
end
{* T2EngineSize: Idem *}

T2AutoSound in Attribute2Def with
  attrProp
    attrASound:CarModel!autoSound
  derivAttr
    derivASound:"multi"
end

T3SpecialEquip in Attribute3Def with
  attrProp
    attrSEquip:CarModel!specialEquip
  genAttrType
    genSEquip: String
```

```
  genAttrPropag
    genPropagSEquip: "T3Inst"
end

String in TypeDef end
```

Classes `CarModel` and `Car` are created as ordinary classes, independently of their participation in a materialization relationship:

```
CarModel in Class with
  attribute
    name:String;
    stickerPrice:Integer;
    #doors:Integer;
    engineSize:Integer;
    autoSound:Integer;
    specialEquip:String
end

Car in Class with
  attribute
    manufDate:Date;
    serial#:Integer;
    owner:String
end
```

To take into account the materialization relationship between them, the two classes are declared as instances of `AClass` and `CClass`, respectively.

During the creation of `CarModel` as an instance of `AClass`, meta-attribute `materializes` of `AClass` is instantiated by `materializesCar`. In attribute `CarModel!materializesCar`, we specify that attributes `name` and `stickerPrice` propagate with `T1`; `#doors` and `engineSize` both propagate with `T2`, and each produces a monovalued instance attribute, while `autoSound` produces, also with `T2`, a multivalued instance attribute; `specialEquip` generates, with `T3`, new instance attributes of type `String`.

7.7.1.2 Constraints Related to Propagated Attributes Two constraints are related to propagated attributes, expressing that propagated attributes are attributes of the abstract classes involved in materialization relationships and that `T2` and `T3` attributes must be `multi`valued.

All propagated attributes appearing in the definition of meta-attribute `AClass!materializes` (presented in section 7.7.1.1) must be attributes of the more abstract

class. For instance, T1 attributes (name, stickerPrice), T2 attributes (#doors, engineSize, and autoSound), and T3 attribute (specialEquip) in the partial implementation of CarModel—*Car presented in section 7.7.1.1 must be attributes of the abstract class CarModel.

The corresponding constraint T1AttrCnstr for T1 attributes is as follows (a constraint for T2 and T3 attributes is similar):

```
AClass!materializes with
  constraint
    T1AttrCnstr:
    $ forall M/AClass!materializes A/AClass C/CClass T1/
    Attribute1Def attr/Attribute
    From(M,A) and To(M,C) and (M inhAttr1 T1) and (T1 attrProp
    attr)
    ==> From(attr,A) $
end
```

The following definition of constraint T2attrAreMultivalued requires that T2 attributes be multivalued:

```
  AClass with
    constraint
      T2attrAreMultivalued:
      $ forall T2/Attribute2Def attr/Attribute
      (T2 attrProp attr)
      ==> (attr in Class!multivalued) $
  end
```

The constraint shows that attribute attr propagating with mode T2 is constrained to be an instance of Class!multivalued. The predefined metaclass Class is extended by meta-attribute multivalued, whose domain is Class itself, as shown here:

```
Class with
  attribute
    multivalued:Class
  constraint
    multiCnstr:
    $ forall p/Class!multivalued s,d/Class i/Proposition l/Label
    P(p,s,l,d) and (i in s)
    ==> (exists d1,d2/Proposition (d1 in d) and (d2 in d) and (i l
    d1) and (i l d2) and not(d1==d2)) $
end
```

The associated constraint `multiCnstr` enforces that every instance `p` of `Class!multivalued` has at least two distinct values, `d1` and `d2`.[3] A similar constraint, say `T3attrAreMultivalued`, can be defined to constrain `T3` attributes to be multivalued.

Note that `Class!multivalued` can be used by any class to express that some of its attributes are multivalued. This is achieved by declaring the attributes in question as instances of `Class!multivalued`.

sources

7.7.2 Instance-Level Semantics

This section defines the required structure and behavior relevant to the instance-level semantics of materialization. Constraints and rules are attached to `AClass` and `CClass` and operate on abstract and concrete objects.

First, recall that according to the semantics of materialization, each abstract object `a`, an instance of a given abstract class `A`, necessarily has a class facet that is a subclass of the concrete class materializing `A`. That information cannot be associated with metaclass `AClass` because it concerns abstract objects and not abstract classes. Therefore, a specific class, say, `AbstractObject`, is needed to supply that information to all abstract objects:

```
AbstractObject in Class with
  attribute, necessary, single
    classFacet: Class
end
```

7.7.2.1 Constraints Related to Abstract Objects Constraint `abstractObjCnstr` expresses that instances of instances of `AClass` must be instances of `AbstractObject`:

```
AClass with
  constraint
    abstractObjCnstr:
    $ forall A/AClass a/Proposition
    (a in A) ==> (a in AbstractObject) $
end
```

For instance, `FiatRetro`, an instance of `CarModel` that is in turn an instance of `AClass`, must be an instance of `AbstractObject`.

Upon creating an abstract object `a` as an instance of `AbstractObject`, the system will necessarily require the instantiation of attribute `classFacet`, since this latter is declared (in the definition of `AbstractObject` in section 7.7.2) as being `necessary` and `single`, meaning that it must have exactly one value, which represents the class facet associated with object `a`. A complementary constraint `objClassFacetCnstr`

requires that this class facet must necessarily be a subclass of the concrete class materializing the class of a:

```
AClass with
  constraint
    objClassFacetCnstr:
    $ forall A/AClass C/CClass a/AbstractObject Cf/Class
    (A materializes C) and (a in A) and (a classFacet Cf) ==> (Cf
    isA C) $
end
```

Thus, according to the constraints abstractObjConstr and objClassFacetConstr, an instance such as FiatRetro of CarModel must be declared as an instance of AbstractObject and must be associated with a class facet, say FiatRetroCars, that must be a subclass of Car materializing CarModel, the class of FiatRetro:

```
FiatRetro in AbstractObject with
  classFacet
    cfFRCars:FiatRetroCars
end

FiatRetroCars isA Car end
```

7.7.2.2 Constraints Related to Concrete Objects Constraint concreteObjCnstr expresses that all concrete objects c, instances of a given concrete class C and materializing a given abstract object a, must be instances of the class facet associated with a:

```
AClass with
  constraint
    concreteObjCnstr:
    $ forall A/AClass C/CClass M/AClass!materializes
    a/AbstractObject Cf/Class c/Proposition
    From(M,A) and To(M,C) and (a in A) and (a classFacet Cf) and
    (Cf isA C) and (c in C) ==> (c in Cf) $
end
```

For example, Nico's Fiat that is an instance of Car and that materializes FiatRetro must necessarily be an instance of FiatRetroCars, the class facet associated with FiatRetro (see the abstract object FiatRetro, defined in section 7.7.2.2). Note that all instances of FiatRetroCars are implicitly instances of Car, because of the isA link, but the inverse is not guaranteed. That is the reason why constraint concreteObjCnstr is explicitly defined.

7.7.2.3 Cardinality Constraints As mentioned previously, concrete classes always have cardinality $(1,1)$ in materializations. That cardinality is expressed by the conjunction of two constraints `minCardCnstrIs1` and `maxCardCnstrIs1`:

```
CClass with
  constraint
    minCardCnstrIs1:
    $ forall C/CClass A/AClass M/AClass!materializes c/Proposition
    From(M,A) and To(M,C) and (c in C)
    ==> exists a/AbstractObject Cf/Class (a in A) and (a
    classFacet Cf) and (Cf isA C)$ ;
    maxCardCnstrIs1:
    $ forall C/CClass A/AClass M/AClass!materializesc/Proposition
    From(M,A) and To(M,C) and (c in C)
    ==> (forall a1,a2/AbstractObject Cf/Class
    (a1 in A) and (a2 in A) and (a1 classFacet Cf) and (a2
    classFacet Cf) and (Cf isA C) ==> (d1 == d2) ) $
end
```

Constraint `minCardCnstrIs1` means that a given concrete object `c` necessarily corresponds to at least one abstract object `a` (i.e., minimal cardinality is `1`), whereas constraint `maxCardCnstrIs1` means that a given concrete object `c` corresponds to at most one abstract object `a` (i.e., maximal cardinality is `1`).

Note that there is no need to define constraints for cardinality $(0,n)$ in abstract classes. This cardinality is the default in ConceptBase; that is, an abstract object `a` may or may not have corresponding concrete objects.

7.7.3 Attribute Propagation

This section deals with the implementation of attribute propagation related to materialization. Unlike those for generalization, the three mechanisms of attribute propagation provided by materialization concern not only the names of attributes and their domains, but also their values.

Consider the example of `CarModel—*Car`, an abstract object `FiatRetro`, an instance of `CarModel`, and its associated class facet `FiatRetroCars`. The constraints related to attribute propagation must ensure that the following requirements are met:

- `T1` attributes (`name:String`, `stickerPrice:Integer`) of `CarModel` must be attributes of `FiatRetroCars`. Furthermore, the value of those attributes in `FiatRetro` (i.e., `"FiatRetro"` and `10.000`, respectively) must be the same for all instances of `FiatRetroCars`.

- T2 attributes (#doors, engineSize, and autoSound) of CarModel must be attributes of FiatRetroCars. Furthermore, the value of #doors and engineSize for instances of FiatRetroCars must be monovalued and must be selected from the possible values of #doors and engineSize in FiatRetro. As for the value of autoSound for instances of FiatRetroCars, it must be multivalued and must be selected as a subset of the corresponding attribute value in FiatRetro.
- Values of the T3 attribute specialEquip in FiatRetro (i.e., airbag, alarm, and cruise) must be created as new attributes of FiatRetroCars whose domain is supplied in advance by the user, here String.

Now we address how T1, T2, and T3 attributes of CarModel can be physically stored in FiatRetroCars. Three alternatives can be considered:

1. Define a rule that systematically makes the class facet FiatRetroCars a subclass of CarModel. The advantage of this approach is that all attributes of CarModel will automatically be stored once and for all in FiatRetroCars. This approach presents a severe disadvantage, however: When declared as a subclass of CarModel, FiatRetroCars inherits not only attributes but also constraints and rules concerning these attributes. For instance, constraint T2attrAreMultivalued (defined in section 7.7.1.2) requires that #doors and engineSize be multivalued for instances of CarModel. If FiatRetroCars is considered a subclass of CarModel, this constraint will be inherited, transparently to the user. As a result, #doors and engineSize will be constrained to be multivalued for instances of FiatRetroCars, which is obviously an undesirable behavior. For this reason, this approach must be discarded.

2. Define an active rule that will systematically add T1, T2, and T3 attributes of CarModel to class facet FiatRetroCars. This active rule will be required to execute after the creation of abstract object FiatRetro and its associated FiatRetroCars, which is just a subclass of Car. Furthermore, some constraints and/or rules will have to be supplied to control the values of the propagated attributes. This is the approach used in the following.

3. Define some constraints that just alert the user that T1, T2, and T3 attributes of CarModel must be made attributes of FiatRetroCars. This approach reveals nothing about the way these attributes are stored in FiatRetroCars. All that is required is that propagated attributes be stored in FiatRetroCars. Thus, if a propagated attribute has not been stored in FiatRetroCars, the associated constraint will request the user to add it explicitly.

7.7.3.1 Propagation of T1 Attributes Active rule T1Propagation automatically propagates T1 attributes of an abstract class A to each class facet associated with each instance of A:

```
ECArule T1Propagation with
  ecarule
    T1AttrInClassFacetRule:
    $ M/AClass!materializes A/AClass C/CClass T1/Attribute1Def
    a/AbstractObject Cf/Class att/Attribute
    ON Tell((a classFacet Cf))
    IF From(M,A) and To(M,C) and (Cf isA C) and (M inhAttrT1 T1)
    and (T1 attrProp att) and (a in A)
    DO CALL(PropagAttribute(A,Cf,att)) $
end
```

Rule `T1AttrInClassFacetRule` reads as follows: Upon the association of a class facet `Cf` with a given abstract object `a`, if `A`, the class of `a`, defines its attribute `att` as a `T1` attribute, then execute the predicate `PropagAttribute(A,Cf,att)`,[4] which will carry out the propagation of `att` from abstract class `A` to class facet `Cf`.

Rule `T1AttrAreClassAttrRule` states that `T1` attributes are *class* attributes:

```
AClass with
  rule
    T1AttrAreClassAttrRule:
    $ forall c,v/Proposition l/Label
    (exists M/AClass!materializes A/AClass C/CClass
    T1/Attribute1Def a/AbstractObject Cf/Class att/Attribute
    From(M,A) and To(M,C) and (M inhAttrT1 T1) and (T1 attrProp
    att) and Label(att,l) and (a in A) and (a classFacet Cf) and
    (a l v) and (c in Cf))
    ==> (c l v) $
end
```

In class facets, for a given `T1` attribute `T1`, if the value of the propagated attribute `att` in a given abstract object `a` is `v`, then all instances `c` of the class facet `Cf` associated with `a` will come with the same value `v`.

Another alternative for dealing with `T1` attributes is the use of a *delegation* mechanism (Lieberman 1986) that would permit concrete instances to access directly the `T1` attribute values in abstract instance `a` without storing it redundantly in class facets. Unfortunately, ConceptBase does not provide facilities for using a delegation mechanism.

7.7.3.2 Propagation of T2 Attributes Active rule `T2MonoPropagation` automatically propagates `T2` monovalued attributes of an abstract class `A` to each class facet associated with each instance of `A`:

```
ECArule T2MonoPropagation with
  ecarule
    T2MonoAttrInClassFacetRule:
    $ M/AClass!materializes A/AClass C/CClass T2/Attribute2Def
    a/AbstractObject Cf/Class att/Attribute
    ON Tell((a classFacet Cf))
    IF From(M,A) and To(M,C) and (Cf isA C) and (M inhAttrT2 T2)
    and (T2 attrProp att) and (T2 derivAttr "mono") and (a in A)
    DO CALL(PropagAttribute(A,Cf,att)),
    TELL((att in Class!single)) $
end
```

Rule `T2MonoAttrInClassFacetRule` reads similarly to rule `T1AttrInClassFacetRule` (presented in section 7.7.3.1). However, in part DO of rule `T2MonoAttrInClassFacetRule`, not only do `T2` attributes `att` propagate from abstract class `A` to class facet `Cf`, but they are also asserted as instances of `Class!single` to express their "monovaluedness." The constraint controlling the values of `T2` attributes is as follows:

```
AClass with
  constraint
    T2ValuePropCnstr:
    $ forall M/AClass!materializes A/AClass C/CClass
    T2/Attribute2Def a/AbstractObject att/Attribute Cf/Class
    c,v/Proposition l/Label
    From(M,A) and To(M,C) and (M inhAttrT2 T2) and (T2 attrProp
    att) and Label(att,l) and (a in A) and (a classFacet Cf) and
    (c l v) and (c in Cf) ==> (a l v) $
end
```

(A similar active rule, say, `T2MultiAttrInClassFacetRule`, can be defined for propagating multivalued `T2` attributes.) Values of `T2` attributes (either monovalued or multivalued) in concrete objects are restricted to a set of fixed values supplied by the corresponding abstract object `a` (see the preceding constraint definition).

Constraint `T2ValuePropCnstr` reads as follows: For each `T2` propagated attribute `att`, labeled `l` in class facet `Cf`, which is associated with abstract object `a`, if a concrete object `c`, an instance of `Cf`, has value `v` for that attribute, then `v` is necessarily a value of that attribute in the corresponding object facet `a`. For instance, the `T2` attribute `#doors:Integer` of `CarModel` must be stored as an attribute of class facet `FiatRetroCars` associated with `FiatRetro`, an instance of `CarModel`. The domain of `#doors` in `FiatRetroCars` is also `Integer`, but its value in `Nico's Fiat`, an

instance of FiatRetroCars, is either 3 or 5, as specified by the abstract instance FiatRetro.

7.7.3.3 Propagation of T3 Attributes Active rule T3Propagation implements the propagation of T3 attributes:

```
ECArule T3Propagation with
  ecarule
    T3ValuePropRule:
    $ M/AClass!materializes A/AClass C/CClass T3/Attribute3Def
    a/AbstractObject Cf/Class att/Attribute v/Proposition
    D/TypeDef l,lab/Label
    ON Tell((a classFacet Cf))
    IF From(M,A) and To(M,C) and (Cf isA C) and (M inhAttrT3 T3)
    and (T3 attrProp att) and Label(att,l) and (T3 genAttrType D)
    and (a in A) and (a l v) and P(v,v,lab,v)
    DO CALL(AddNewAttribute(Cf,lab,D) $
end
```

Rule T3ValuePropRule deals with the association of a class facet Cf with a given abstract object a. If A, the class of a, defines att as a T3 attribute with domain D for the generated attributes, and if v, labeled lab, is a possible value of att for abstract object a, then predicate AddNewAttribute(Cf,lab,D) is executed,[5] which creates a new attribute with label lab and domain D in class facet Cf.

Consider the example of CarModel—*Car. After the creation of abstract object FiatRetro of CarModel and of its associated class facet FiatRetroCars, then, upon the instantiation of its T3 attribute specialEquip with a set of values airbag, alarm, and cruise, the active rule automatically inserts airbag, alarm, and cruise as new attributes of FiatRetroCars with domain String specified in field genAttrType of structure Attribute3Def (see the partial implementation of CarModel—*Car presented in section 7.7.1.1).

7.7.4 Multiple Materializations

The generic semantics of simple materializations presented in the previous sections also hold for multiple materializations in which an abstract class materializes in more than one concrete class (e.g., CDBookCopy*—Book—*BookCopy). Let A be an abstract class that materializes in two concrete classes C_1 and C_2 (i.e., C_1*—A—*C_2). The following declarations show how this case is subsumed by the generic definitions previously given.

Classes A, C_1, and C_2 are first declared independently of materialization. Then to take materializations C_1*—A—*C_2 into account, A is created as an instance

of AClass, and attribute materializes of AClass is instantiated in A by two attributes, materializesC$_1$ and materializesC$_2$, with destinations C$_1$ and C$_2$, respectively:

```
A in AClass with
  materializes
    materializesC1:C1;
    materializesC2:C2
end

C1 in CClass end
C2 in CClass end

A!materializesC1 with
  inhAttrT1 ...
  inhAttrT2 ...
  inhAttrT3 ...
end
{* Idem for A!materializesC2 *}
```

Then, meta-attributes A!materializesC$_1$ and A!materializesC$_2$ are declared as instances of metaclass Class to capture different characteristics of materializations C$_1$*—A and A—*C$_2$, respectively. In such materializations, an abstract object a, an instance of A, must have two associated class facets Cf$_1$ and Cf$_2$ as subclasses of C$_1$ and C$_2$, respectively. This requirement is ensured by the constraint ObjClassFacetConstr for simple materializations defined in section 7.7.2.1.

7.7.5 Composition of Materializations

This section shows how compositions of materializations such as Play—*Setting—*Performance are implemented using the definitions given in the previous sections. Let A be an abstract class that materializes in concrete class C that in turn materializes in D (i.e., A—*C—*D). In such a composition of materializations, the intermediate class C is at the same time concrete in materialization A—*C and abstract in C—*D. Therefore, whereas A and D are to be declared as instances of AClass and CClass, respectively, C has to be declared as an instance of both AClass and CClass. As ConceptBase allows multiple classifications, this requirement is easily implemented:

```
A in AClass with
  materializes
    materializesC:C
end
```

```
C in AClass, CClass with
  materializes
    materializesD:D
end

D in CClass end

A!materializesC with
  inhAttrT1 ...
  inhAttrT2 ...
  inhAttrT3 ...
end
{* Idem for C!materializesD *}
```

Now let us see how attributes propagate in composition of materializations. `T1`, `T2`, and `T3` attributes of class `C` propagate to class `D` exactly as in any simple materialization. As for `T1`, `T2`, and `T3` attributes of the *root* abstract class of `A—*C—*D` (i.e., `A`), they propagate to the leaf concrete class `D` as follows. Let `a` be an instance of `A`, `Cf_a` its associated class facet (which is a subclass of `C`); `c` an instance of `Cf_a` (indirectly an instance of `C`), `Cf_c` its associated class facet (which is a subclass of `D`); and finally `d` an instance of `D`. Instances `d` of `D` have attribute values corresponding to all attributes defined in `A`. Only those attributes of `A` propagating with `T3` and `genAttrPropag = T3-T2mono` or `genAttrPropag = T3-T2multi` should be physically stored in `Cf_c`. `T1` and `T2` attributes as well as `T3` attributes with `genAttrPropag = T3Inst` are also physically stored in `Cf_c`, but in data models supporting delegation, there is no need to store them in `Cf_c`.

7.7.6 Querying Materializations

Generic queries can be defined against materialization relationships. These queries may include the following:

- Give all concrete classes for a given abstract class.
- Give the abstract class for a given concrete class.
- Given two classes `A` and `C`, test whether `A` is an abstract class of `C`.
- Given a materialization `A—*C`, return all `T1` (or `T2` and `T3`) attributes of `A`.
- Give all concrete objects (instances of a given concrete class) materializing a given abstract object.
- Give the abstract object for a given concrete object.

For example, query `AllCClasses` returns all concrete classes associated with a given abstract class `A` passed as parameter:

```
AllCClasses in GenericQueryClass isA CClass with
  parameter
    A: AClass
  constraint c: $ (A materializes this)$
end
```

For the partial instantiation of `CarModel—*Car` presented in section 7.7.1.1, the answer to `AllCClasses` with `CarModel` as parameter would be `Car`. Remember that the answer (object) we look for is indicated by the predefined operator `this`.

Another generic query, `AllCObjects`, at the instance level is as follows:

```
AllCObjects in GenericQueryClass with
  parameter
    abstobj:AbstractObject;
    concrclass:CClass
  Constraint
    c: $ exists Cf/Class (abstobj classFacet Cf) and (Cf isA
    concrclass) and (this in Cf)$
end
```

Query `AllCObjects` returns all concrete objects associated with a given abstract object `abstobj` passed as parameter. The concrete class `concrclass` materializing the class of `abstobj` must also be passed as parameter. For the abstract object `FiatRetro` defined in section 7.7.2.2, the answer to `AllCObjects` with `FiatRetro` as parameter `abstobj` and class `Car` as parameter `concrclass` consists of `Nico's Fiat`.

7.8 Conclusion

This chapter has first presented the materialization data abstraction. Materialization is a generic relationship between a conceptual class and its concrete manifestations. Specifically, we have given a formal definition of materialization as a combination of the usual generalization and classification generic relationships and a class/metaclass correspondence; we have characterized several mechanisms of attribute propagation through materialization; we have exhibited examples that demonstrate that materialization is frequently encountered in practice; and we have surveyed related work that involved concepts similar to materialization.

In addition, the chapter has presented an implementation of materialization in ConceptBase. Two metaclasses, `AClass` and `CClass`, were built as templates to capture the semantics of materialization at the class level, and an additional class `AbstractObject` was defined to capture some particular semantics of materialization that only concern the instance level.

Metaclass `AClass` was used to define meta-attribute `materializes`, which refers to metaclass `CClass`. Thanks to the class status of ConceptBase attributes, the meta-attribute `materializes` can be declared as a metaclass to carry the semantics of materialization. A set of attributes were attached to meta-attribute (metaclass) `materializes` to represent the structural semantics of materialization, such as the three associated propagation types. A collection of roughly one dozen rules and constraints were defined to ensure the behavioral semantics of materialization at both the class and the instance levels in a coordinated manner.

Specific materializations were created as instances of the meta-attribute `materializes` and the abstract and concrete classes involved in them were created as instances of the metaclasses `AClass` and `CClass`, respectively. Upon these instantiations, the system then automatically enforced the corresponding constraints and rules derived from the metalevel. Compositions of materializations in which a concrete class is also an abstract class in another materialization were treated by means of multiple classifications.

Our implementation has demonstrated the power and flexibility of ConceptBase for integrating new generic relationships like materialization. Here are in a nutshell the strengths of ConceptBase that played a central role in our implementation:

- the first-class status of attributes
- an unlimited extensibility by metaclass hierarchies with the possibility of multiple classification
- the ability to specify deductive rules and integrity constraints as ordinary attributes of (meta)classes
- the fact that queries are ordinary classes with membership constraints
- active rules that allow the specification of arbitrary actions to be executed if certain events/conditions hold in the object base

Although the expressive power of these last is still limited, their presence is vital. We used them, in particular, to implement the three attribute propagation modes associated with materialization.

Our implementation has also pointed out, however, some limitations of ConceptBase:

- *Parameterized queries within constraints*: Some definitions of constraints and rules seem to be complex and difficult to read. This would not be the case if ConceptBase made it possible to include parameterized queries in the definition of constraints. This facility would allow expressions like `inst in ClassA` and `inst in QueryB[list of instantiated parameters]` to be written within a constraint of a class.
- *Dynamically propagating/adding an attribute to a given class*: To define the attribute/value propagation mechanisms of materialization, specific predicates like

those proposed in section 7.7.3 are needed to dynamically propagate attributes from source to destination classes.

• *The* `IF` *clause of active rules*: The definition of active rules supports only one expression in their `IF` clauses. This is certainly too restrictive. In fact, some complex active rules, such as the active rule for propagating `T1` attributes in section 7.7.3.1, that for propagating monovalued `T2` attributes in section 7.7.3.2, and that for propagating `T3` attributes in section 7.7.3.3, naturally require more than one expression in their `IF` clauses. Complex conditions in active rules are allowed in ConceptBase, but only one free variable is permitted. Using `Ask` statements in the `DO` part can circumvent this.

Notes

1. The notion of abstractness/concreteness of materialization captures domain semantics. It is distinct from the notion of abstract class in object models, in which an abstract class is a class without instances, whose complete definition is typically deferred to subclasses.

2. For the sake of clarity, this domain is not shown in figures 7.12 and 7.13.

3. Meta-attribute `multivalued` is defined in the same spirit as meta-attributes `single` and `necessary` (see chapter 3). Note also that a multivalued attribute can be viewed as a `necessary` and `not single` attribute.

4. The semantics of the predicate `PropagAttribute(C1,C2,attr)` are more generic. They consist in copying (and not moving) an attribute `attr` from a source class `C1` to a target class `C2`. Attribute `attr` is assumed to belong to class `C1` but not to `C2`.

5. The semantics of predicate `AddNewAttribute(C,Lab,Dest)` are more generic. They add a new attribute with label `Lab` and destination (domain) `Dest` in class `C`.

References

Al-Jadir, L., T. Estier, G. Falquet, and M. Léonard. 1995. "Evolution Features of the F2 OODBMS." In *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)*, ed. T. W. Ling and Y. Masunaga, 284–291. Singapore: World Scientific.

Andonoff, E., G. Hubert, A. Parc, and G. Zurfluh. 1996. "Integrating Versions in the OMT Models." In *Proceedings of the Fifteenth International Conference on Conceptual Modeling (ER'96)* (Lecture Notes in Computer Science 1157), ed. B. Thalheim, 472–487. Berlin: Springer-Verlag.

Bertino, E. 1992. "A View Mechanism for Object-Oriented Databases." In *Proceedings of the Third International Conference on Extending Database Technology (EDBT'92)* (Lecture Notes in Computer Science 779), ed. A. Pirotte, C. Delobel, and G. Gottlob, 136–151. Berlin: Springer-Verlag.

Coad, P., D. North, and M. Mayfield. 1995. *Object Models: Strategies, Patterns, and Applications*. Upper Saddle River, NJ: Yourdon.

Dahchour, M. 1998. "Formalizing Materialization Using a Metaclass Approach." In *Proceedings of the Tenth International Conference on Advanced Information Systems Engineering (CAiSE'98)* (Lecture Notes in Computer Science 1413), ed. B. Pernici and C. Thanos, 401–421. Berlin: Springer-Verlag.

Dahchour, M. 2001. "Integrating Generic Relationships into Object Models Using Metaclasses." Ph.D. diss., Department of Computing Science and Engineering (INGI), University of Louvain, Louvain, Belgium.

Dahchour, M., A. Pirotte, and E. Zimányi. 2002a. "Materialization and Its Metaclass Implementation." *IEEE Transactions on Knowledge and Data Engineering* 14, no. 5: 1078–1094.

Dahchour, M., A. Pirotte, and E. Zimányi. 2002b. "A Generic Role Model for Dynamic Objects." In *Proceedings of the Fourteenth International Conference on Advanced Information Systems Engineering (CAiSE*

2002) (Lecture Notes in Computer Science 2348), Toronto, ed. A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. Tamer Ozsu, 643–658. New York: Springer-Verlag.

Dahchour, M., A. Pirotte, and E. Zimányi. 2004. "A Role Model and Its Metaclass Implementation." *Information Systems* 29, no. 3: 235–270.

Dahchour, M., A. Pirotte, and E. Zimányi. 2005. "Generic Relationships in Information Modeling." *Journal of Data Semantics* 4 (Lecture Notes in Computer Science 3730), ed. S. Spaccapietra, 1–34. New York: Springer.

Falquet, G., M. Léonard, and J. Sindayamaze. 1994. "F2-Concept: A Database System for Managing Classes Extensions and Intensions." In *Information Modeling and Knowledge Bases V*, ed. H. Jaakola, H. Kangassalo, T. Kitahashi, and A. Márkus, 243–256. Amsterdam: IOS Press.

Fowler, M. 1997. *Analysis Patterns: Reusable Object Models*. Reading, MA: Addison-Wesley.

Goldstein, R. C., and V. C. Storey. 1994. "Materialization." *IEEE Transactions Knowledge and Data Engineering* 6, no. 5: 835–842.

Gottlob, G., M. Schrefl, and B. Röck. 1996. "Extending Object-Oriented Systems with Roles." *ACM Transactions on Office Information Systems* 14, no. 3: 268–296.

Halper, M., J. Geller, and Y. Perl. 1998. "An OODB Part-Whole Model: Semantics, Notation, and Implementation." *Data & Knowledge Engineering* 27, no. 1: 59–95.

Hay, D. 1996. *Data Models Patterns: Conventions of Thought*. New York: Dorset House.

Johnson, R., and B. Woolf. 1998. "Type Object." In *Pattern Languages of Program Design*, vol. 3, ed. R. Martin, D. Riehle, and F. Buschmann. Reading, MA: Addison-Wesley.

Kilov, H., and J. Ross. 1994. *Information Modeling: An Object-Oriented Approach*. Upper Saddle River, NJ: Prentice Hall.

Klas, W., and M. Schrefl. 1995. *Metaclasses and Their Application*. Lecture Notes in Computer Science 943. New York: Springer-Verlag.

Kolp, M. 1999. "A Metaobject Protocol for Integrating Full-Fledged Relationships into Reflective Systems." Ph.D. diss., INFODOC, Université Libre de Bruxelles.

Martin, J., and J. Odell. 1995. *Object-Oriented Methods: A Foundation*. Upper Saddle River, NJ: Prentice Hall.

Motschnig-Pitrik, R., and J. Mylopoulos. 1996. "Semantics, Features, and Applications of the Viewpoint Abstraction." In *Proceedings of the Eighth International Conference on Advanced Information Systems Engineering (CAiSE'96)* (Lecture Notes in Computer Science 1080), Crete, ed. P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, 514–539. New York: Springer-Verlag.

Motschnig-Pitrik, R., and V. C. Storey. 1995. "Modeling of Set Membership: The Notion and the Issues." *Data & Knowledge Engineering* 16, no. 2: 147–185.

Mylopoulos, J. 1998. "Information Modeling in the Time of the Revolution." *Information Systems* 23, nos. 3–4: 127–155.

Pirotte, A., E. Zimányi, D. Massart, and T. Yakusheva. 1994. "Materialization: A Powerful and Ubiquitous Abstraction Pattern." In *Proceedings of the Twentieth International Conference on Very Large Data Bases (VLDB'94)*, ed. J. Bocca, M. Jarke, and C. Zaniolo, 630–641. San Francisco: Morgan Kaufmann.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design*. Upper Saddle River, NJ: Prentice Hall.

Tabourier, Y. 1997. "Les power types ont 20 ans." *Ingénierie des Systèmes d'Information* 5, no. 5: 15–30.

Wieringa, R. J., W. De Jonge, and P. Spruit. 1995. "Using Dynamic Classes and Role Classes to Model Object Migration." *Theory and Practice of Object Systems* 1, no. 1: 61–83.

Yang, O., M. Halper, J. Geller, and Y. Perl. 1994. "The OODB Ownership Relationship." In *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'94)*, London, ed. D. Patel, Y. Sun, and S. Patel, 278–291. New York: Springer-Verlag.

Zimányi, E. 1997. "Implementing Materialization in Logtalk." YEROOS technical report no. TR-97/09, Laboratoire de Bases de Données, Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, Switzerland.

8 Metadatabase Design for Data Warehouses

Christoph Quix

8.1 Introduction

Data warehouses (DWs) provide large-scale caches of historical data. They sit between information sources gained externally or through online transaction processing (OLTP) systems and decision support or data-mining systems, following the vision of online analytic processing (OLAP). There are three main arguments in favor of the caching approach that DWs represent:

1. *Performance and safety considerations*: The concurrency control methods of most database management systems (DBMSs) do not react well to a mix of short update transactions (as in OLTP) and OLAP queries, which typically search a large portion of the database. Moreover, OLTP systems are often critical for the operation of an organization and must not be under danger of interference from other applications.

2. *Logical interpretability problems*: Inspired by the success of spreadsheet techniques, OLAP users tend to think in terms of highly structured, multidimensional data models, whereas information sources offer at best relational, often just semistructured data models.

3. *Temporal and granularity mismatch*: OLTP systems focus on current operational support in great detail, whereas OLAP often considers historical developments at a somewhat less detailed granularity.

Thus, quality considerations have accompanied data warehouse research from the beginning. A large body of literature has evolved over the past few years in regard to addressing the problems introduced by the DW approach, such as the trade-off between timeliness of DW data and disturbance of OLTP work during data extraction, the minimization of data transfer through incremental view maintenance, and a theory of computation with multidimensional data models.

However, the frequent use of highly qualified consultants in data warehouse applications indicates that we are far from a systematic understanding and usage of the

interplay between quality factors and design options in data warehousing. The goal of the European DWQ project (Jarke and Vassiliou 1997) was to address these issues by developing, prototyping, and evaluating comprehensive "foundations for data warehouse quality," delivered through *enriched metadata management facilities* in which specific analysis and optimization techniques are embedded.

Discussions with DW tool vendors, DW application developers, and DW administrators has shown that the standard framework used in the DW literature is insufficient to capture in particular the business role of data warehousing. A DW is a major investment made to satisfy some business goal of an enterprise; the quality model selected and the DW design should reflect this business goal as well as its subsequent evolution over time. The next section discusses this problem in detail. The new architectural framework I present in that section separates (and links) explicitly the concerns of conceptual enterprise perspectives, logical data modeling (the main emphasis of DW research to date), and physical information flow (the main concern of commercial DW products to date).

8.2 An Extended Data Warehouse Architecture

The traditional data warehouse architecture, recommended both in research and in the commercial trade press, is shown in figure 8.1. Physically, a data warehouse system consists of databases (source databases, materialized views in the data warehouse); data transport agents, which ship data from one database to another; and a repository that stores metadata about the system and its evolution. In this architecture, heterogeneous information sources are first made accessible in a uniform way through extraction mechanisms called *wrappers*, then *mediators* (Wiederhold 1992) take on the task of integrating information from different data sources and resolving conflicts between inconsistent information in the data sources. The resulting standardized and integrated data are stored as materialized views in the data warehouse. The DW base views are usually just slightly aggregated; in order to customize them for different groups of analyst users, *data marts* containing more-aggregated data about specific domains of interest are frequently constructed as second-level caches. These caches are then accessed by data analysis tools ranging from query facilities and spreadsheet tools to full-fledged data-mining systems based on knowledge-based or neural-network techniques.

The content of the repository determines to a large extent the way the data warehouse system can be used and evolved. The main goal of the approach described in this section is therefore to define a metadatabase schema that can capture and link all relevant aspects of DW architecture and quality.

I tackle this difficult task in several steps. First, I present standards for metadata management in data warehousing and discuss their shortcomings. Then, I describe an example from industrial practice that shows the complexity of data warehouse

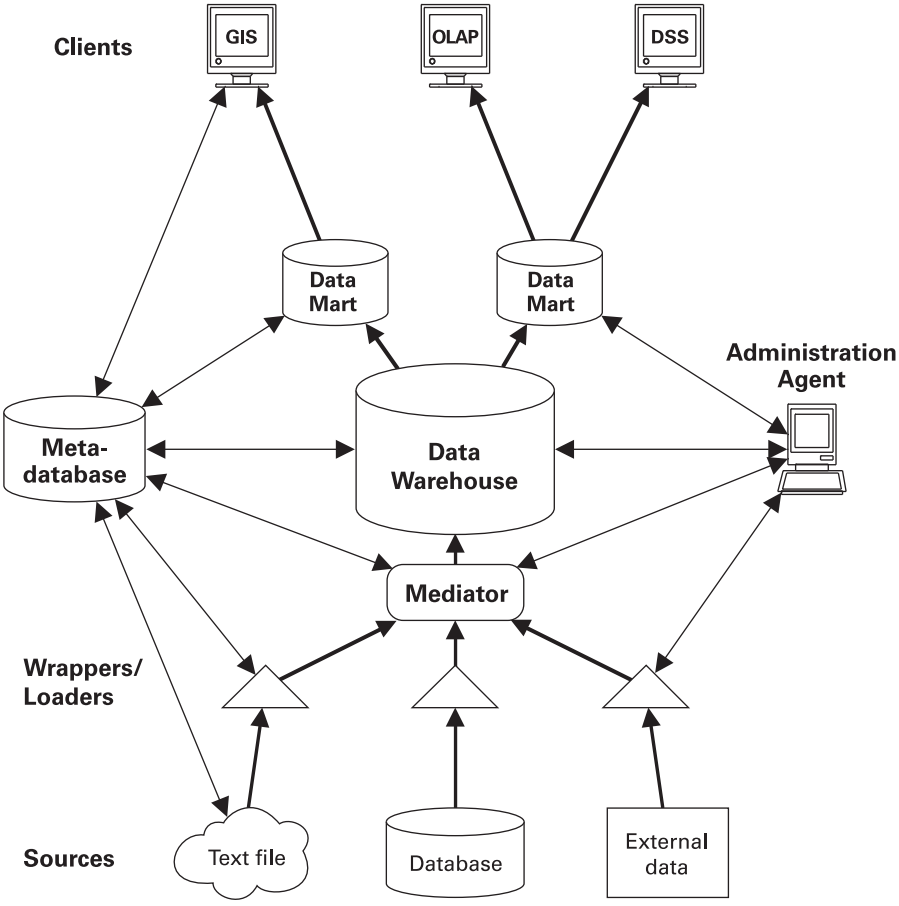


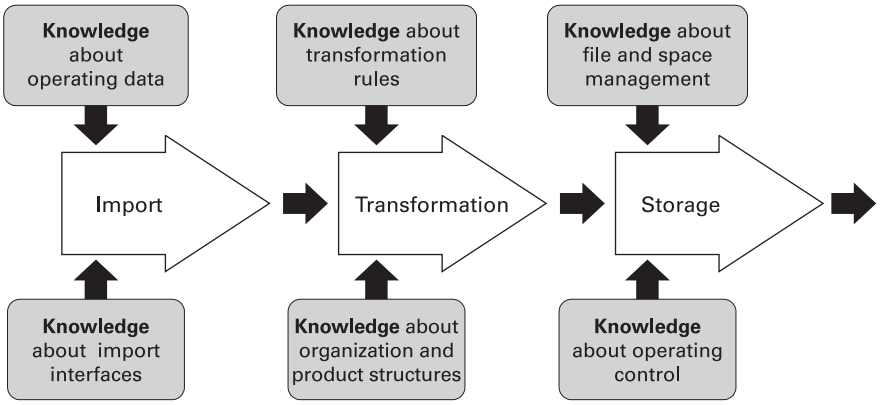
Figure 8.1
Traditional data warehouse architecture

metadata. Next, I discuss the shortcomings of the traditional architecture and propose a conceptual enterprise perspective to solve some of these shortcomings. Finally, I elaborate on the extended metamodel resulting from the approach presented in this section and show how it can be implemented in a repository.

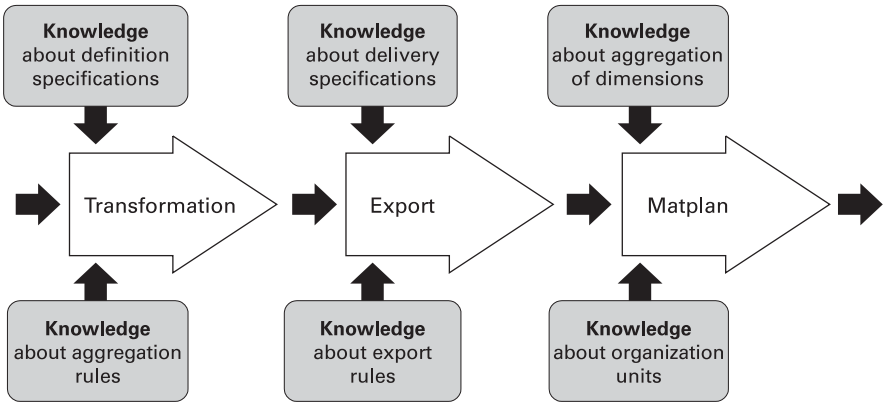
8.2.1 Standards for Data Warehouse Metadata

Two industry standards have emerged in recent years: the Open Information Model (OIM) by the Metadata Coalition (MDC), and the Common Warehouse Metamodel (CWM) by the Object Management Group (OMG).[1] A detailed comparison of both approaches can be found in Vaduva and Vetterli 2001 and Vetterli, Vaduva, and Staudt 2000.

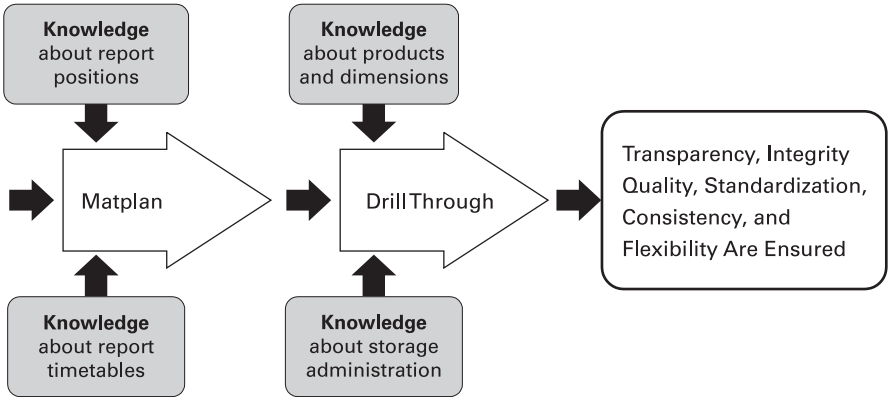
(a)


Knowledge about operating data
Knowledge about transformation rules
Knowledge about file and space management
Import
Transformation
Storage
Knowledge about import interfaces
Knowledge about organization and product structures
Knowledge about operating control

(b)


Knowledge about definition specifications
Knowledge about delivery specifications
Knowledge about aggregation of dimensions
Transformation
Export
Matplan
Knowledge about aggregation rules
Knowledge about export rules
Knowledge about organization units

(c)


Knowledge about report positions
Knowledge about products and dimensions
Matplan
Drill Through
Transparency, Integrity Quality, Standardization, Consistency, and Flexibility Are Ensured
Knowledge about report timetables
Knowledge about storage administration

In essence, the goal of OIM is to support life-cycle-wide tool interoperability. The portion of OIM focused on data warehousing addresses the description of static aspects such as database schema elements (following the SQL standard), OLAP schema elements, and record-oriented database elements and report definitions, as well as a hierarchy of dynamic aspects, including data transformation maps, transformation steps, and transformation packages. OIM uses UML both as a modeling language and as the basis for its core model. It is divided into submodels, or *packages*, that extend UML in order to address the different areas of information management. The *Database and Warehousing Model* is composed of the *Database Schema Elements* package, the *Data Transformations Elements* package, the *OLAP Schema Elements* package, and the *Record Oriented Legacy Databases* package. The Database Schema Elements package contains three other packages: a *Schema Elements* package (covering the classes, modeling tables, views, queries, indexes, etc.), a *Catalog and Connections* package (covering physical properties of a database and the administration of database connections), and a *Data Types* package (standardizing a core set of database data types).

CWM is narrower but provides more support for data warehousing itself. In addition, it differs from OIM by relying on object-oriented and semistructured modeling technologies such as UML, XML, and CORBA (Common Object Request Broker Architecture). It is organized in a foundational metamodel comprising business information, data types and CWM types, expressions, keys, and indexes, on which a number of more specific model packages for both MOLAP (Multidimensional On-Line Analytical Processing) and ROLAP (Relational On-Line Analytical Processing) solutions are based. These address warehouse deployment in terms of hardware and software; packages for accessing relational, XML-based, and record-oriented sources; packages to enable multidimensional and relational data warehouses to reach OLAP functionality; and process-oriented packages comprising individual transformations, process flow, and day-to-day operation.

The shortcomings of these standards have been discussed in (Vetterli, Vaduva, and Staudt 2000). Both standards focus mainly on technical metadata, and neither provides detailed mechanisms for handling business metadata. Moreover, neither standard has any support for capturing information about the quality of data in a data warehouse. Therefore, we want an extended metadata model for data warehouses to capture not only technical metadata but also business metadata. The next subsection motivates the approach presented in this section by means of an example from industrial practice.

◀ **Figure 8.2**
Metadata sources in the data warehouse control process of a bank (Schäfer et al. 2000): (a) loading and refreshing the operational data store; (b) preparing a large number of specialized mutlidimensional models; (c) personalizing information delivery and providing access to background information

8.2.2 The Need for Extended Metadata Management

The challenge of metadata management is probably best illustrated by figure 8.2, which describes the basic information flow in DB-Prism, a data warehouse for financial control in a large international bank (Schäfer et al. 2000). The figure lists no less than sixteen submodels addressing different perspectives of metadata, reflecting the difficulties of reconciling different kinds of financial semantics, heterogeneity of source data models and source availability, bottlenecks involved in scheduling huge data flows during daily refreshment, and personalization of client interests beyond their role definitions.

A conceptual perspective documenting the meaning of data and their relationships has been a critical success factor of the system. However, the mapping of this conceptual approach to the logical level—that is, the incremental creation of data in the warehouse from various sources and the restructuring of data into numerous cube formats and report structures—has been equally important. Given the size and complexity of the system, it is not surprising that physical-level optimization is also very important.

These aspects cannot be considered independently of one another but are closely interlinked. The precise documentation and, where possible, automation of these tasks is the goal of the metadata management facilities in DB-Prism. Therefore, it is necessary to develop an integrated metadata framework that captures all relevant aspects of data warehouse metadata.

8.2.3 Adding a Conceptual Perspective to Data Warehousing

Almost all current research and practice consider a data warehouse architecture as a stepwise information flow from information sources through materialized views toward analyst clients, as shown in figure 8.1. For example, projects such as The Stanford-IBM Manager of Multiple Information Sources (TSIMMIS) (Chawathe et al. 1994), Squirrel (Hull and Zhou 1996), and Ware House Information Prototype at Stanford (WHIPS) (Hammer et al. 1995) focus on the integration of heterogeneous data via wrappers and mediators, using different logical formalisms and technical implementation techniques. The *Information Manifold* project at AT&T Research (Levy, Srivastava, and Kirk 1995) is the only one that provides a conceptual domain model as a basis for integration. A key observation is that the architecture in figure 8.1 covers the tasks faced in data warehousing only partially and is therefore unable even to express, let alone support, a large number of important data quality problems and strategies for the management of data.

The main argument I wish to make is the need for a *conceptual enterprise perspective*. In figure 8.3, the flow of information introduced in figure 8.1 is stylized on the right-hand side, whereas the process of creating and using the information is shown

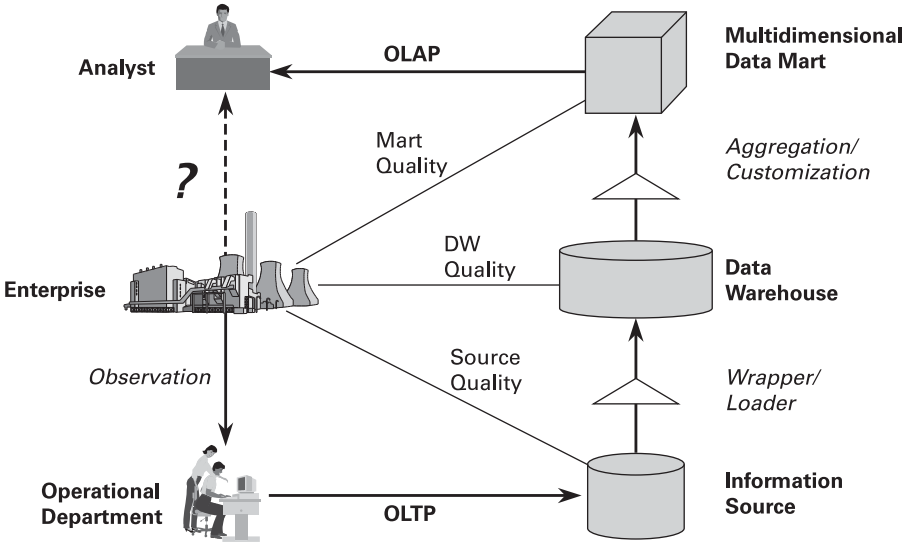


Figure 8.3
Data warehousing in the context of an enterprise

on the left. Suppose an analyst wants to know something about the business—the question mark in the figure. He or she does not have the time to observe the business directly but must rely on existing information gained by operational departments, which is documented as a side effect of OLTP systems. This way of gathering information implies a bias that needs to be compensated for when OLTP data are selected for uploading into a DW and cleaning; there they are then further preprocessed and aggregated in data marts for certain analysis tasks. Considering the long path the data have taken, it is obvious that the last step, the formulation of queries that are conceptually adequate for the information needs of the business analyst and the conceptually adequate interpretation of the answers, also presents a major problem to the analyst.

The traditional DW literature covers only two of the five steps depicted in figure 8.3. Thus, it has no answers to typical practical questions such as "How can my operational departments put so much money into their data quality, and still the quality of my DW is terrible?" (answer: The enterprise views of the operational departments are not easily compatible with one another or with the analysts' view) or "What is the effort required to analyze problem x for which the DW currently offers no information?" (answer: It could simply be a problem of inappropriate aggregation in the materialized views or it could require access to not-yet-integrated OLTP sources, or it might even involve setting up new OLTP sensors in the organization).

An adequate answer to such questions requires an explicit model of the conceptual relationships between an enterprise model, the information captured by OLTP departments, and the OLAP clients whose task is the decision analysis. I have argued that a DW is a major investment made for a particular business purpose. Therefore the enterprise model is not a minor part of the environment but instead it is required that *all other models be defined as views on this enterprise model*. Perhaps surprisingly, even information source schemas define views on the enterprise model—not vice versa, as suggested by figure 8.1.

8.2.4 A Repository Model for the Extended Data Warehouse Architecture

Through the introduction of an explicit business perspective as shown in figure 8.3, the wrapping and aggregation transformations performed in the traditional data warehouse literature can all be checked for interpretability, consistency, and completeness with respect to the enterprise model, provided an adequately powerful representation and reasoning mechanism is available. At the same time, the logical transformations need to be implemented safely and efficiently through physical data storage and transportation—the third perspective in the approach described in this section. It is clear that these physical quality aspects require completely different modeling formalisms than the conceptual ones; typical techniques stem from queuing theory and combinatorial optimization. As a consequence, the data warehouse metaframework proposed clearly separates three perspectives, as shown in figure 8.4: a

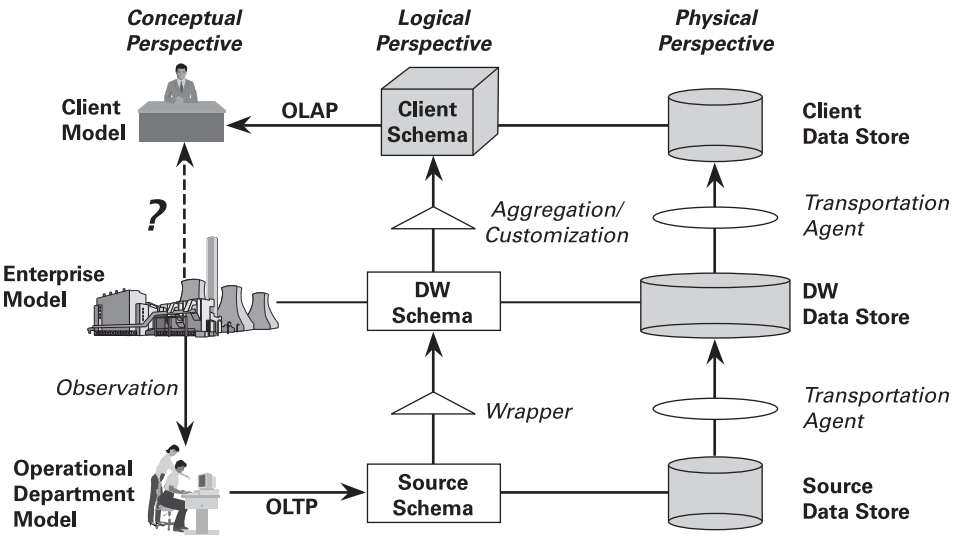


Figure 8.4
The proposed data warehouse metadata framework

conceptual enterprise perspective, a logical data-modeling perspective, and a physical data flow perspective.

No single decidable formalism could handle all these aspects uniformly in a metadatabase. Instead, the architectural framework is captured in a deductive object data model in a comprehensive but relatively shallow manner. Special-purpose reasoning mechanisms such as the ones mentioned previously can be linked to the architectural framework as it is discussed in section 8.3.

The metadatabase system ConceptBase and its modeling language Telos are used to store an abstract representation of data warehouse applications in terms of the three-perspective scheme just outlined. The advantage of ConceptBase is that it provides query facilities and definitions of constraints and deductive rules, thus enabling analysis and consistency checking of models. Telos is well suited to analysis of models because it allows specialized modeling notations (including the adaptation of graphical representations [Jarke et al. 1999]) to be formalized by means of metaclasses. Since ConceptBase treats all concepts including metaclasses as first-class objects, it is suitable for managing abstract representations of DW objects to be measured (Jeusfeld et al. 1998).

sources

A condensed ConceptBase model of the architecture notation is given in figure 8.5, using the graph syntax of Telos. Heavy arrows denote specialization links. The top

gel

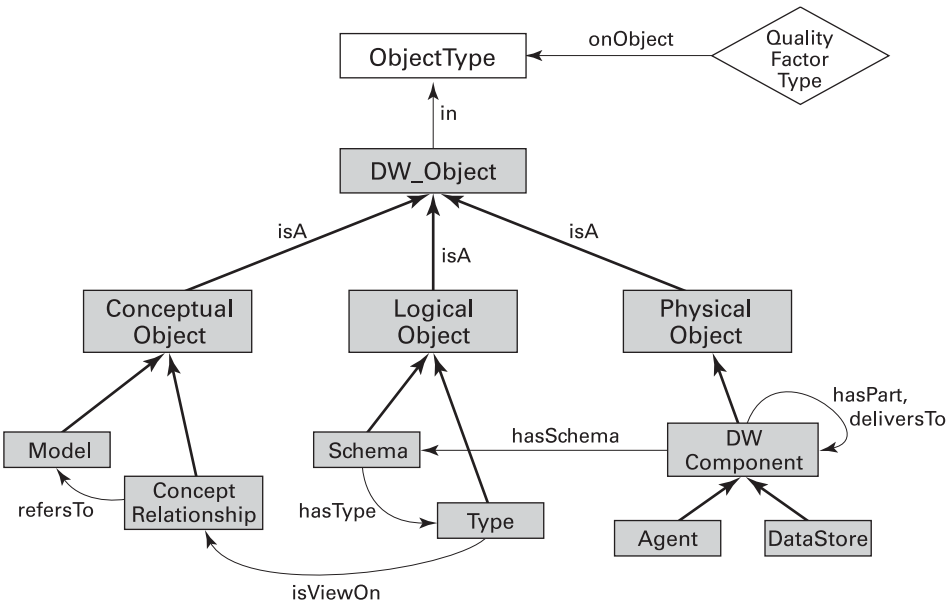


Figure 8.5
Structure of the repository metamodel

level object is `ObjectType`. It is a meta-metaclass that is instantiated by all metaclasses of the model. `DW_Object` is the base class for all metaclasses related to the architecture of a data warehouse. It classifies objects from any perspective (conceptual, logical, or physical) and at any level (source, data warehouse, or client). Within each perspective, I distinguish between the modules it offers (e.g., client model) and the type of information found within these modules (e.g., concepts and their subsumption relationships). The horizontal links `hasSchema` and `isViewOn` establish the way the horizontal links in figure 8.4 are interpreted: The types of schemas (i.e., relational or multidimensional structures) are defined as logical views on the concepts in the conceptual perspectives. On the other hand, the components of the physical perspective get a schema from the logical perspective.

Each object in the metamodel for data warehouses can have an associated set of materialized views called *quality factors*. These materialized views (which can also be specialized to the different perspectives [not shown in the figure]) constitute the bridge to the quality model. Thus, a link is established between the `QualityFactorTypes` (i.e., classes of quality factors) and the object types. This link is discussed in more detail in section 8.3.

Experimentation with this notation has shown that it can represent physical data warehouse architectures of commercial applications. The logical perspective currently supports relational schema definitions, whereas the conceptual perspective supports the family of extended entity-relationship and similar semantic data-modeling languages. Note that all objects in figure 8.5 are metaclasses: actual conceptual models, logical schemas, and data warehouse components are represented as instances of the metaclasses in the metadatabase. In the following subsections, I elaborate on the purpose of representing each of the three perspectives.

8.2.5 Conceptual Perspective

The conceptual perspective offers a business model of the information systems of an enterprise. The central role is played by the enterprise model, which gives an integrative overview of the conceptual objects of an enterprise. The models of both the client and the source information systems are views on the enterprise model; that is, their contents are described in terms of the enterprise model. This is the major difference between my approach and the classical approach in data warehouse systems, in which the ''global'' enterprise model is usually seen as a view on the local source models. One goal of the conceptual perspective is to provide a model of the information that is independent of the physical organization of the data, so that relationships between concepts can be analyzed by intelligent tools in order to simplify the integration of the information sources, for example. On the client side, the interests of user groups can also be described as views on the enterprise model.

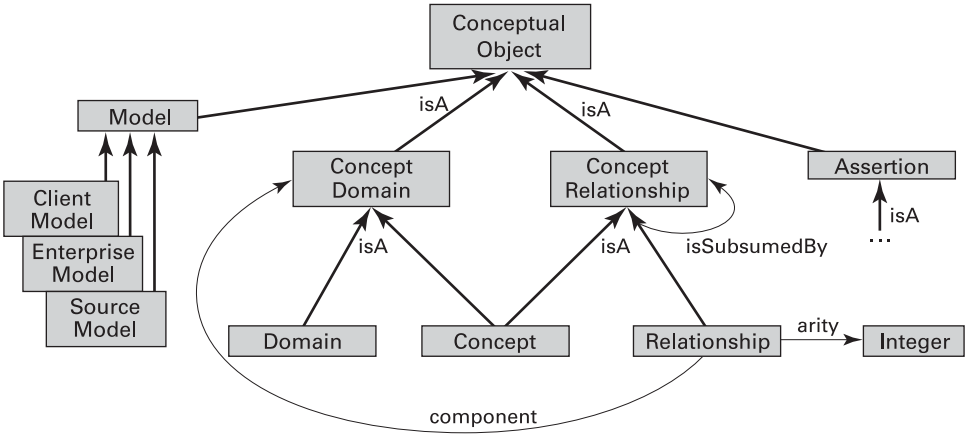


Figure 8.6
Part of the metamodel in the conceptual perspective

In the implementation of the conceptual perspective in the metadatabase, the central class is called `Model` (see figure 8.6). A `Model` is related to a source, to a client, or to the relevant section of the enterprise, and it represents the concepts that are available in the corresponding source, client, or enterprise. The classes `ClientModel`, `SourceModel`, and `EnterpriseModel` are needed to distinguish among the models of several sources, clients, and the enterprise itself. A model consists of `Concepts` and `Relationships`. A `Concept` represents a concept of the real world, that is, the business world. `Relationships` are *n*-ary relationships between concepts or domains (e.g., integer or string). For example, "employee" and "department" can be represented as concepts, and "works in" is a binary relationship between employee and department. Basically, this implementation allows the representation of extended entity-relationship models. Furthermore, semantic relationships between concepts can be expressed as `Assertions` in a formal language, that is, a description logic (Calvanese et al. 2001). These assertions can be formulated to express generic domain knowledge (`DomainAssertions`), properties and limitations of a source (`IntraModelAssertions`), and relationships between sources, such as containment and consistency (`InterModelAssertions`). This allows a reasoner to check the consistency and correctness of the conceptual model and to check for the subsumption of concepts. This information can then be used to determine which sources need to be accessed for the materialization of a particular concept in the data warehouse.

The results of the reasoning process are stored in the model as attributes `isSubsumedBy` of the corresponding concepts. Essentially, the repository can serve as a cache for reasoning results. Any tool can ask the repository whether or not a

particular concept is subsumed by another concept. If the result has already been computed, the query can be answered directly by the repository. Otherwise, a reasoner is invoked by the repository to compute the result.

The metadata repository ConceptBase also provides some basic functionality for checking the correctness of the conceptual model. The following example shows the definition of the class `Relationship`. As indicated in figure 8.6, a relationship has an arity and relates a number of concepts or domains. Constraints that are checked by the metadata repository are used to ensure that a relationship relates the correct number of concepts and each concept has a unique position in the relationship:

```
Class Relationship in ObjectType isA ConceptRelationship with
  attribute,single,necessary
    arity : Integer
  attribute
    component : ConceptDomain
  constraint
    arity_constraint :
    $ forall r/Relationship i/Integer
    (r arity i) ==> (i >= 2 ) $;
  position_constraint :
    $ forall r/Relationship c1,c2/Relationship!component a/Integer
    p1,p2/Integer
    (Ai(r,component,c1) and Ai(r,component,c2) and (c1 position
    p1) and (c2 position p2) and not(c1 == c2) and A(r,arity,a))
    ==> (not (p1 == p2) and (p1 <= a) and (p2 <= a)) $;
  all_positions_constraint :
    $ forall r/Relationship i/Integer a/Integer
    A(r,arity,a) and (i <= a) ==> (exists c/Relationship!component
    Ai(r,component,c) and (c position i)) $
end
```

So far, I have shown how the models of the sources and the enterprise are represented in the conceptual perspective. I now briefly describe the conceptual model at the client level. Data warehouse systems use special multidimensional structures to represent their data in a way that is most efficient for the user. Therefore, in the conceptual client model, it is important to know how aggregations are defined and which attributes of a concept are aggregated. Figure 8.7 shows the client level of the metamodel for the conceptual perspective.

`Aggregations` aggregate concepts with respect to a specific `DimensionLevel`, which is defined by a `DimensionAttribute` and a `Level`. For example, if customers are aggregated by cities, the dimension attribute is "address" and the level is "city."

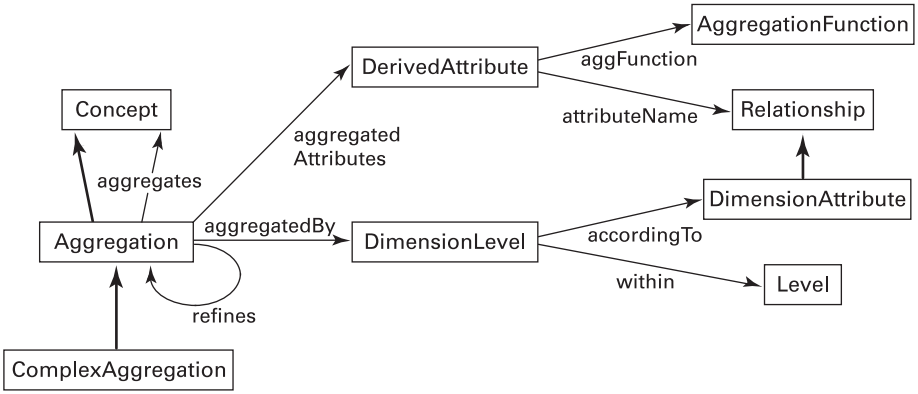


Figure 8.7
Conceptual model for the client level

Furthermore, we need to know which attributes are aggregated and which `AggregationFunction` is used for the aggregation.

8.2.6 Logical Perspective

The logical perspective conceives a data warehouse from the viewpoint of the actual data models involved; that is, the data model of the logical schema is given by the corresponding physical component that implements the logical schema. The key concept in the logical perspective is `Schema`. As a model consists of concepts, a schema consists of `Types`. The relational model was implemented as an example of a logical data model; other data models such as the multidimensional or the object-oriented data model can also be integrated into this framework (Gebhardt, Jarke, and Jacobs 1997; Vassiliadis 1998).

As in the conceptual perspective, I distinguish in the logical perspective between `ClientSchema`, `DWSchema`, and `SourceSchema` for the schemata of clients, the data warehouse, and the sources. For each client or source model, there is one corresponding schema. This restriction is guaranteed by a constraint in the architecture model. The link to the conceptual model is implemented through the relationship between concepts and types: Each type is expressed as a view on concepts.

The metamodel in the logical perspective can be further refined to capture the definition of queries that are used to extract the data from the sources (see Jarke et al. 1999 for details).

8.2.7 Physical Perspective

The data warehouse industry has mostly explored the physical perspective, so many aspects in the physical perspective are taken from the analysis of commercial data

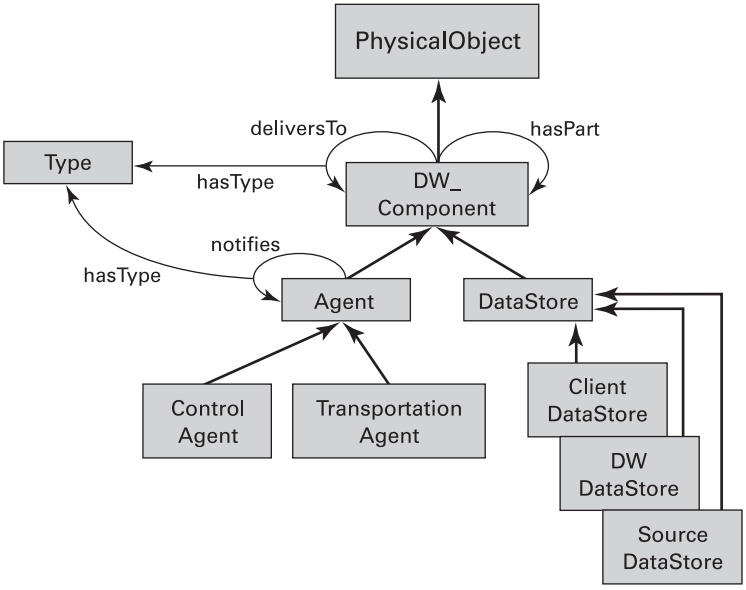


Figure 8.8
Physical perspective of the data warehouse metamodel

warehouse solutions and tools, such as the product suite of MicroStrategy (www.microstrategy.com), the Oracle Warehouse Builder tools (www.oracle.com), and the Extract-Transform-Load (ETL) tools of Ascential Software (www.ascentialsoftware.com) and Informatica (www.informatica.com). The basic physical components in a data warehouse architecture are `Agents` and `DataStores`. `Agents` are programs that control other components or transport data from one physical location to another. `DataStores` are databases, which store the data that are delivered by other components.

As shown in figure 8.8, the basic class in the physical perspective is `DW_Component`. A data warehouse component may be composed of other components, as is expressed by the attribute `hasPart`. Furthermore, a component `deliversTo` another component a `Type`, which is part of the logical perspective. Another link to the logical model is the attribute `hasSchema` of `DW_Component`. Note that a component may have a schema, that is, a set of several types, but it can deliver only a type to another component. This is a result of the observation that agents usually transport only "one tuple at a time" of a source relation rather than a complex object.

There are two types of `Agents`: `ControlAgents`, which control other components and agents (e.g., `notify` another agent to start the update process), and `TransportationAgents`, which transport data from one component to another component. An `Agent` may also notify other `Agents` about errors or its termination.

A `DataStore` physically stores the data that are described by models and schemata in the conceptual and logical perspective. As in the other perspectives, I distinguish between `ClientDataStore`, `DW_DataStore`, and `SourceDataStore` for data stores of clients, the data warehouse, and the sources, respectively.

8.3 Managing Data Warehouse Quality

In this section, I discuss how to extend the DW architecture model to support explicit quality models. There are two basic issues to be resolved. On the one hand, quality is a subjective phenomenon, so quality goals must be organized according to the stakeholder groups that pursue these goals. On the other hand, quality goals are highly diverse. They can be neither assessed nor achieved directly but require complex measurement, prediction, and design techniques, often in the form of an interactive process. The overall problem of introducing quality models in metadata is therefore how to achieve wide coverage without giving up the detailed knowledge available for certain criteria. Only a combination of these two elements enables systematic quality management.

The following subsections show how a basic structure of quality dimensions can be formally captured in an extension to the goal-question-metric approach from software engineering and how such an extension can be implemented and used in the DW metadatabase. A detailed discussion of quality dimensions for data warehouses can be found in Jarke et al. 1999 and Quix et al. 1999.

8.3.1 Hierarchical Quality Assessment: An Adapted Goal-Question-Metric Approach

It is clear that there can be no decidable formal framework that comes close to covering all aspects of quality measurement in a uniform language. When designing the metadatabase extensions for quality management, another solution that still maintains the overall picture offered by shallow quality management techniques such as quality function deployment (QFD) but is at the same time is open to the embedding of specialized assessment and design techniques is required.

The solution to this problem proposed in this section builds on the goal-question-metric (GQM) approach widely used in software quality management (Oivo and Basili 1992). The idea of GQM is that quality *goals* can usually not be assessed directly. Instead, their meaning is circumscribed by *questions* that need to be answered when evaluating the quality. Quality questions again can usually not be answered directly but rely on *metrics* applied to either the product or process in question; techniques such as statistical process control charts are then applied to derive, from the measurements, the answer for a particular question.

In the preceding example, the goal of responsiveness can be refined into questions about the trade-off between query and update performance (logical perspective),

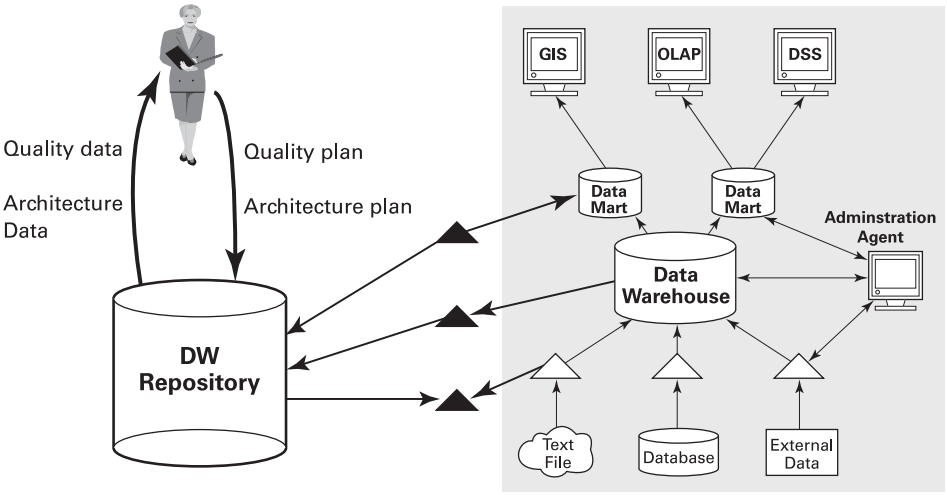


Figure 8.9
Quality management via the data warehouse repository

about the bottlenecks present at the physical level, and about the completeness or even redundancy of the utilized data sources (conceptual perspective). These questions can then be answered using the previously mentioned metrics.

The repository solution uses a similar approach to bridge the gap between quality goal hierarchies on the one hand and very detailed metrics and reasoning techniques on the other. The bridge is defined by the idea of quality queries as materialized views over the data warehouse; the views are defined by generic queries over the quality measurements. Figure 8.9 motivates this approach by zooming in on the repository. The stakeholder assesses the quality of the data warehouse by posing queries concerning quality to the repository. The repository answers the queries by accessing quality data obtained from measurement agents (the black triangles in figure 8.9). The agents communicate with the components of the real data warehouse to extract measurements.

The stakeholder may redefine his or her quality goals at any time. This leads to an update of the quality model in the repository and possibly to the configuration of new measurement agents responsible for delivering the base quality data. Analogously, a stakeholder with appropriate authorization can redefine the architecture of the data warehouse via the repository. Such an evolutionary update (e.g., the specification of a new data source) leads to a reconfiguration of the real data warehouse. Ultimately, the quality measurements will then reflect an effect of such an update and will give evidence as to whether the evolution has led to an improvement in any of the quality goals.

The use of the repository for data warehouse quality management has significant advantages:

- Data warehouse systems already incorporate repositories to manage metadata about the data warehouse; extending this component for quality management is a natural step.
- Existing metadata about the data warehouse (e.g., source schemas) can be directly used for formulating quality goals and measurement plans.
- The quality model can be kept consistent with the architecture model (i.e., the repository can prevent the stakeholders from formulating quality goals that cannot be validated with the given architectural data).
- The stakeholder accesses the repository as a *data source* for delivering quality reports to the stakeholders, who formulate quality goals; in fact, producing such reports is the same kind of activity that is performed to deliver aggregated data to the client tools of a data warehouse.

The final advantage is not just a technical remark. Quality data (i.e., values of quality measurements) are derived from DW components. The values are materialized views on the properties of these components. These values have themselves quality properties, such as timeliness and accuracy. It makes a difference whether the value of a quality measurement is updated each hour or once a month. Although I do not go into detail regarding this "second-level" quality, I note that the same methods that are used to maintain the quality of the DW can also be used to maintain the quality of the DW repository (hosting the quality model).

8.3.2 The Quality Metamodel

Quality data are derived data and are maintained by the data warehouse system. The strategy for implementing the quality model in the DW repository provides more technical support than GQM implementations for general software systems. Such systems lack a built-in repository. The expressive query language offered by the ConceptBase repository system simplifies the quality management tasks and supports the DW administrator in monitoring the quality of the DW. In the following, I elaborate on how a version of GQM can be modeled by Telos metaclasses in ConceptBase and then be used for quality goal formulation and quality analysis.

Data warehouse systems are unique in the sense that they rely on a run time metadatabase (or repository) that stores information about the data and processes of the system. This opens the opportunity to implement the GQM approach in such a way that it directly refers to the concepts in the metadatabase of the data warehouse. Under such an implementation, the stakeholders can represent their quality goals explicitly, and the metadatabase maintains the relationship between data warehouse

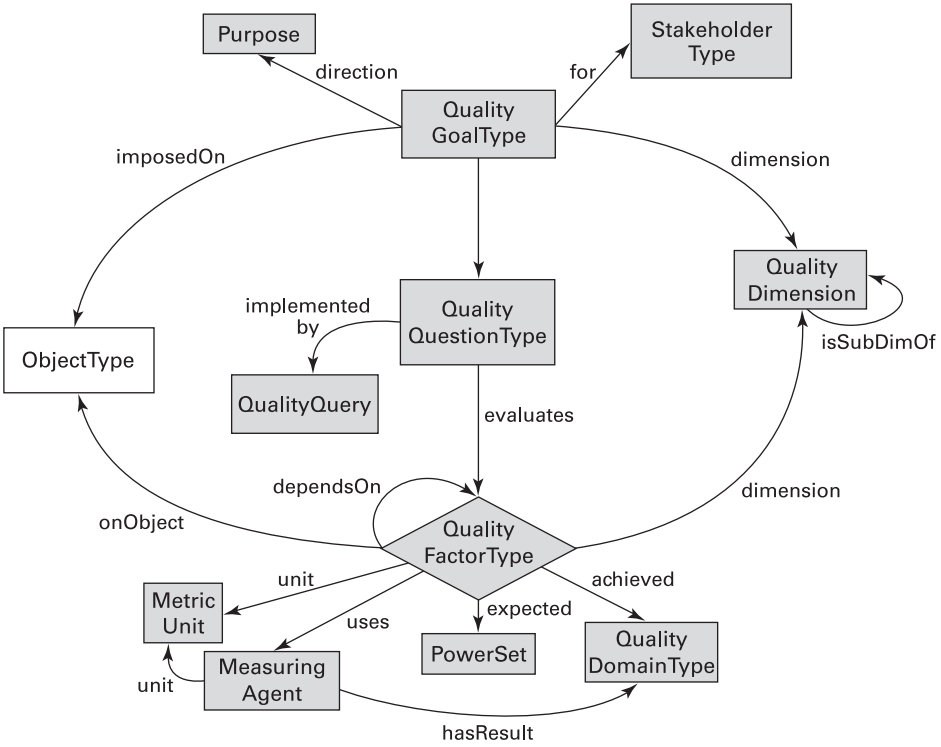


Figure 8.10
A metamodel for data warehouse quality

objects and quality values. Thus, the quality metamodel can be used for both design and analysis purposes. To do so, one has to take into account that a quality factor maps an arbitrary object of a data warehouse (e.g., the number of null values of source relation "Sales") to some value. Thus, a quality factor relates concepts with different abstraction levels, here, a schema concept to a number.

Because of this, the concepts of the quality metamodel (see figure 8.10) are all at the metaclass level (with the notable exception of `ObjectType`, a meta-metaclass). Remember that `ObjectType` has already been used as the base class in the meta-model for DW architecture. The quality metamodel provides the notation for formulating quality goals, queries, and measurements. The quality meta-model will be instantiated in two steps. First, an administrator or a designer of the data warehouse will define types of quality goals and quality factors. For example, "improve the availability of a data store" is a type for a quality goal; that is, it represents a class of quality goals and has to be established for a concrete data store. On the other

hand, "number of minutes offline per week" is a type for a quality factor, which can be measured for a data store. In the second step of the implementation, these types are instantiated by concrete quality goals that have been established for a certain data warehouse object or by concrete quality factors that represent an actual measurement of a certain data warehouse object. (This is discussed in more detail in section 8.3.3, and an example is given in section 8.4.)

In figure 8.10, a metamodel for data warehouse quality implementing the GQM approach is shown. The upper part of the model allows stakeholders to formulate quality requirements for an object type. The `Purpose` of a `QualityGoal` is interpreted as a `direction` (e.g., "improve" or "achieve" some quality goal). Quality goals are established for a `Stakeholder`. Finally, a quality goal is linked to a `QualityDimension` (e.g., availability, usability, or correctness).

`QualityGoals` are mapped to a collection of `QualityQuestions` that are used to decide whether the goals have been achieved. These questions are implemented as queries to the DW repository. The most simple kind of quality query just evaluates whether the current `QualityMeasurement` for a particular data warehouse object is within the expected interval. A quality measurement uses a metric unit (e.g., the average number of null values per tuple of a relation).

8.3.3 Implementation Support for the Quality Metamodel

The abstraction levels of the concepts in the quality model require closer consideration (Jeusfeld et al. 1998). In standard software metrics, a *quality factor* is a function that maps a real-world entity to a value of a domain, usually a number. In the approach discussed in this section, abstract representations of real-world entities in the DW repository are maintained. Thus, quality factors can be recorded as explicit relationships between the abstract representations (i.e., the DW objects) and the quality values. This is also the reason why the quality factor is represented as a diamond node in figure 8.10, as it represents a relationship between DW objects and quality values. By its nature, such a quality factor relates objects of different abstraction levels. For example, a quality value of 0.8 could be measured for the percentage of null values of the `Employee` relation of some data source. `Employee` is a relation (the type of instances of the `Employee` data structure), whereas 0.8 is just a number.

A remark has to be made on the use of the quality model by instantiation. Typical instances of `ObjectType` are items such as `Relation` (logical perspective) or `Concept` (conceptual perspective). These items are independent of the DW application domain. They are used to describe a DW architecture, but they are not components of a concrete DW architecture. A concrete architecture consists of items such as the data source for `Employee` and concrete wrapper agents. Therefore, when we instantiate the quality model, we describe types of quality goals, types of queries, and types

of measurements. For example, we can describe a completeness goal for relational data sources, which is measured by counting the percentage of null values in the relation. Such types (or patterns) can be reused for any concrete DW architecture. For example, the quality factor for a relational source for `Employee` would be instantiated from the quality factor type by instantiating the expected and achieved quality values. The classification of quality factors is described in detail in Jarke et al. 1999 and Quix et al. 1999.

This two-step instantiation is essential since it allows the repository to be preloaded with quality goal, query, and measurement types independent of the application domain. In other words, the repository has knowledge of quality management methods.

8.4 Example Scenario

The example scenario is based on a case study with Telecom Italia, one of the industrial partners in the DWQ project (Trisolini, Lenzerini, and Nardi 1999). The scenario is as follows: Telecom wants to build a data warehouse that collects information about customers, services, and promotions (indicated by the conceptual enterprise model `TelecomModel`). The data for the warehouse can be integrated from three different sources: the billing department, the statistics department, and the marketing department. Each of the sources has only a part of the information that is necessary for the data warehouse. For example, the billing department has only information about customers and services. Figure 8.11 gives an overview of the conceptual models and also presents a part of the logical schema of the data warehouse. Figure 8.12 shows the physical perspective of the data warehouse.

In the scenario, it is assumed that the users of the warehouse have established a quality goal to achieve more-current data in the warehouse. According to the two-phase instantiation process, we first define a quality goal type:

```
QualityGoalType AchieveMoreCurrentData with
  description
    description : "Not more than a certain percentage of data may
    become invalid due to age"
  direction
    dir : Achieve
  imposedOn
    imposedOn : DataStore
  forPerson
    forPerson : DecisionMaker
  dimension
```

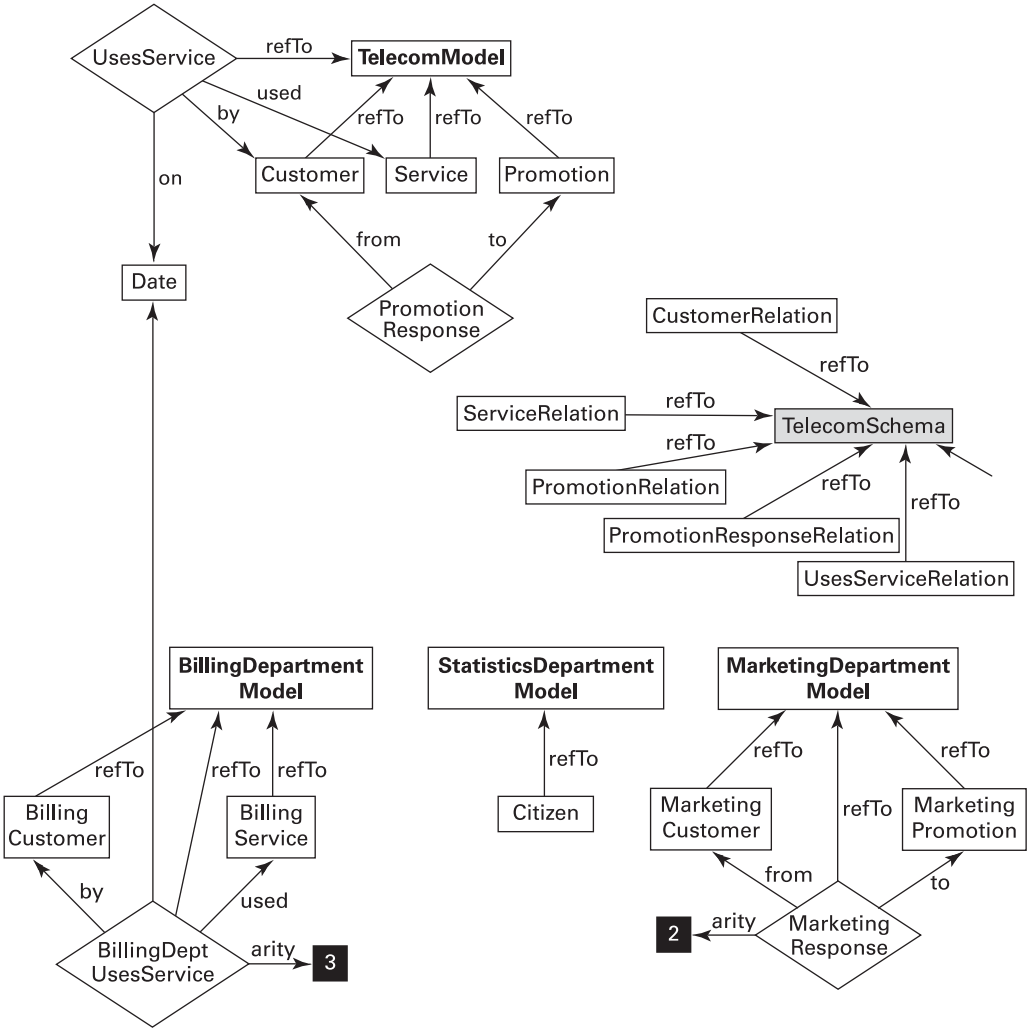


Figure 8.11
Conceptual and logical model of the example scenario (figure 8.12 continues this figure on the right side)

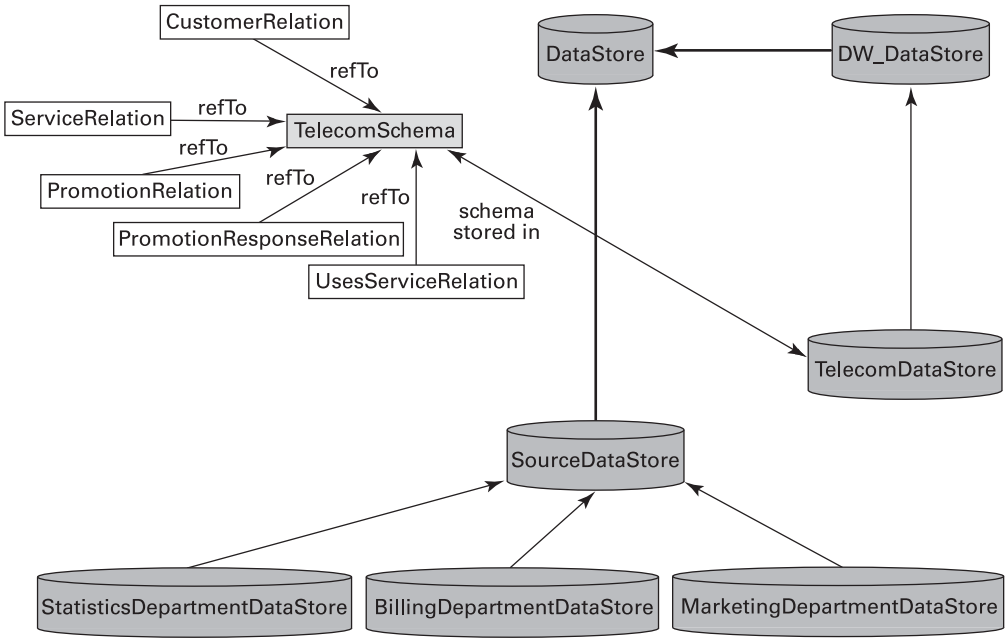


Figure 8.12
Physical perspective of the example (figure 8.11 continues this figure on the left side)

```
    dim : Volatility
  concreteBy
    concreteBy : "Which Data Stores have a volatility outside the
    expected range?"
end
```

Now a decision maker can establish this quality goal for a specific data store. In the example, `"Ron Sommer"` will establish the quality goal on the `TelecomDataStore`:

```
AchieveMoreCurrentData VolatilityForDW with
  imposedOn
    obj : TelecomDataStore
  forPerson
    person : "Ron Sommer"
end
```

The definition of the quality goal type specifies that the goal is concrete by means of a question: `"Which Data Stores have a volatility outside the expected`

range?" The next frames show the definition of this question in Telos and its implementation as a quality query:

```
"Which Data Stores have a volatility outside the expected range?"
in QualityQuestionType with
  implementedBy
    implementedBy : BadVolatilityOfDataStores
  evaluates
    eval : DataStoreVolatility0
end

View BadVolatilityOfDataStores in QualityQuery isA
DataStoreVolatility0 with
  constraint
    c: $ exists i/Integer s/"P[0;100]" (this achieved i) and (this
       expected s) and not (i in s) $
end
```

The definition of the entire quality goal is graphically summarized in figure 8.13. The different abstraction levels are indicated by different shades of gray.

Now the administrator of the data warehouse can check whether the current data warehouse fulfills the requirement of having a volatility inside the expected range by executing the quality query on the metadata repository (see figure 8.14).

The quality query evaluates all quality factors of the type `DataStore-Volatility0` and checks whether the value achieved by the data store is within the expected range. Thus, we can use both the quality and the architecture information of the data warehouse to find weaknesses in the design of the data warehouse. Furthermore, quality factors are linked by a `dependsOn` relationship. This relationship makes it possible for us to trace quality problems back to their sources. To follow the `dependsOn` relationship between quality factors, we could use the following query:

```
QualityQuery CauseOfBadQuality isA DW_Object with
  parameter
    badObject : DW_Object
  constraint
    c: $ exists q1,q2/DataStoreVolatility0 (q1 onObject badObject)
       and (q1 in BadVolatilityOfDataStores) and (q1 dependsOn q2)
       and (q2 in BadQuality) and ((q2 onObject this) or (exists
       o/DW_Object (q2 onObject o) and (this in
       CauseOfBadQuality[o/badObject]))) $
end
```

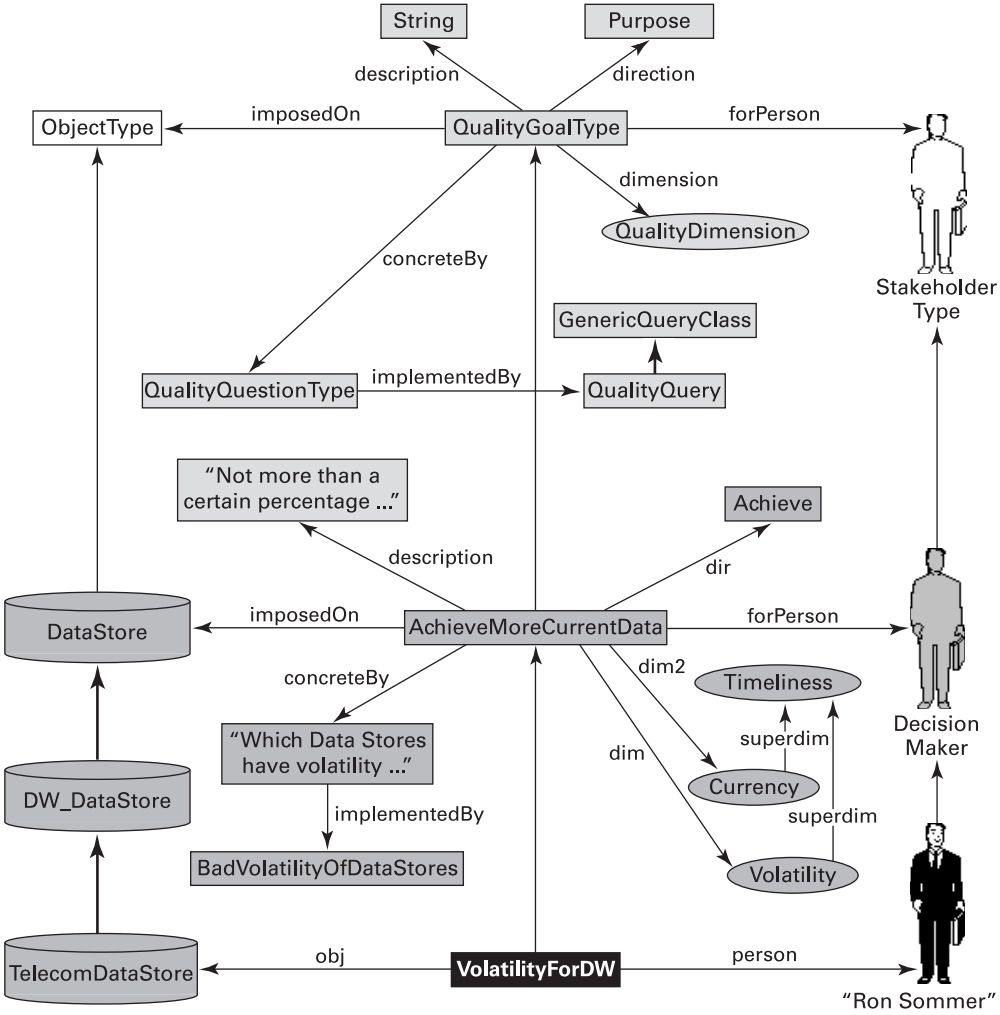


Figure 8.13
Example of a quality goal

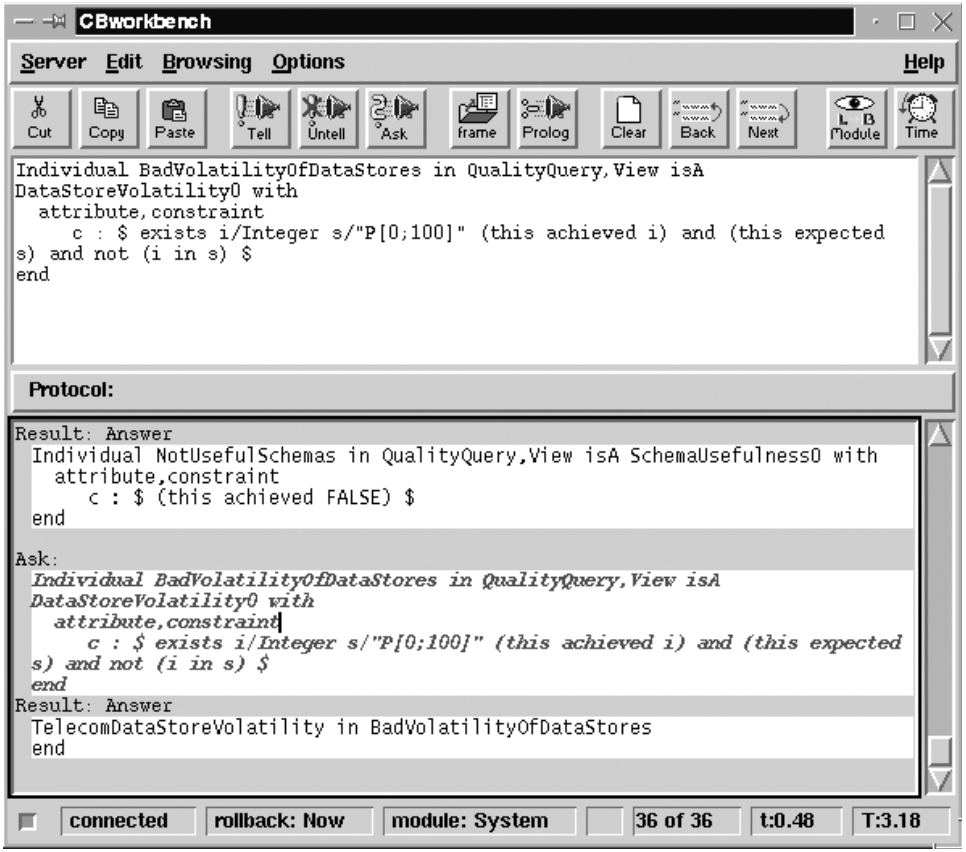


Figure 8.14
A quality query and its result

8.5 Conclusion

The goal of the work I have presented in this chapter is to enrich metadata management in data warehouses such that it can serve as a meaningful basis for systematic quality analysis and quality-driven design. To reach this goal, it was necessary to overcome two limitations of current data warehouse research.

First, the basic architecture in which data warehouses are typically described turned out to be too weak to allow a meaningful assessment of quality of data. As quality is usually detected only by its absence, quality-oriented metadata management requires that we address the full sequence of steps from the capture of the enterprise reality in operational departments to the interpretation of DW information by the client analyst. This again implied the introduction of an explicit enterprise

perspective as a central feature in the architecture. To forestall the possible objection that full enterprise modeling has proven a risky and expensive effort, I remind readers that the approach to enterprise model formation taken in this chapter is fully incremental, so that it is perfectly feasible to construct the enterprise model step by step (e.g., as a side effect of source integration or of other business process analysis efforts).

The second major limitation is the enormous variety of quality factors, each associated with its own measurement and design techniques. The quest for an open quality management environment that could accommodate existing or new techniques of this kind led to an adaptation and repository integration of the GQM approach, in which parameterized queries and materialized quality views serve as the missing link between specialized techniques and the general quality framework.

The power of the repository modeling language determines the boundary between precise but narrow metrics and a comprehensive but shallow global repository. The deductive object base formalism of the Telos language provides a fairly sophisticated level of global quality analysis for prototype implementation but is still fully adaptable and general. Once the quality framework has sufficiently stabilized, a procedurally object-oriented approach could do even more by encoding some metrics directly as methods, (at the expense of flexibility, of course). Conversely, a simple relational metadatabase could take up some of the present models with fewer semantics than offered in the ConceptBase system, but with the same flexibility.

As shown throughout the chapter, this approach has been fully implemented, and some validation has taken place to fine-tune the models. The experiences gained so far indicate that the approach is a promising way toward more systematic and computer-supported quality management in data warehouse design and operation.

Note

1. In 2001, the Metadata Coalition joined with the OMG to develop a common standard for data warehouse metadata. The home page of the Metadata Coalition (www.mdcinfo.com) now redirects users to the home page of the OMG (www.omg.org).

References

Calvanese, D., G. De Giacomo, M. Lenzerini, D. Nardi, and R. Rosati. 2001. "Data Integration in Data Warehousing." *International Journal of Cooperative Information Systems* 10, no. 3: 237–271.

Chawathe, S., H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. 1994. "The TSIMMIS Project: Integration of Heterogeneous Information Sources." In *Tenth Meeting of the Information Processing Society of Japan (IPSJ)*, Tokyo, 7–18. Available at ⟨http://dbpubs.stanford.edu:8090/aux/index-en.html⟩.

Gebhardt, M., M. Jarke, and S. Jacobs. 1997. "A Toolkit for Negotiation Support Interfaces to Multidimensional Data." In *Proceedings of the ACM SIGMOD Conference on Management of Data*, ed. J. Peckham, 348–356. New York: ACM Press.

Hammer, J., H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. 1995. "The Stanford Data Warehousing Project." *IEEE Data Engineering Bulletin*, Special Issue on Materialized Views and Data Warehousing, 18, no. 2: 41–48.

Hull, R., and G. Zhou. 1996. "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, ed. H. Jagadish and I. Singh Mumick, 481–492. New York: ACM Press.

Jarke, M., M. A. Jeusfeld, C. Quix, and P. Vassiliadis. 1999. "Architecture and Quality for Data Warehouses: An Extended Repository Approach." *Information Systems* 24, no. 3: 229–253.

Jarke, M., and Y. Vassiliou. 1997. "Data Warehouse Quality: A Review of the DWQ Project." In *Proceedings of the Second Conference on Information Quality*, ed. D. Strong and B. Kahn, 297–313. Cambridge, MA: MIT Press.

Jeusfeld, M. A., M. Jarke, H. W. Nissen, and M. Staudt. 1998. "ConceptBase: Managing Conceptual Models about Information Systems." In *Handbook of Information Systems*, ed. P. Bernus, K. Mertins, and G. Schmidt, 265–285. Berlin: Springer-Verlag.

Levy, A. Y., D. Srivastava, and T. Kirk. 1995. "Data Model and Query Evaluation in Global Information Systems." *Journal of Intelligent Information Systems* 5, no. 2: 121–143.

Oivo, M., and V. Basili. 1992. "Representing Software Engineering Models: The TAME Goal-Oriented Approach." *IEEE Transactions on Software Engineering* 18, no. 10: 886–898.

Quix, C., P. Vassiliadis, M. Bouzeghoub, and M. Jarke. 1999. "Quality-Oriented Data Warehouse Design." Technical report, DWQ Project. Available at ⟨http://www.dbnet.ece.ntua.gr/~dwq/⟩.

Schäfer, E., J.-D. Becker, A. Boehmer, and M. Jarke. 2000. "Controlling Data Warehouses with Knowledge Networks." In *Proceedings of Twenty-Sixth International Conference on Very Large Data Bases (VLDB)*, ed. A. El Abbadi, M. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter and K.-Y. Whang, 715–718. San Francisco: Morgan Kaufmann.

Trisolini, S., M. Lenzerini, and D. Nardi. 1999. "Data Integration and Warehousing in Telecom Italia." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, ed. A. Delis, C. Faloutsos and S. Ghandeharizadeh, 538–539. New York: ACM Press.

Vaduva, A., and T. Vetterli. 2001. "Meta Data Management for Data Warehousing: An Overview." *International Journal of Cooperative Information Systems* 10, no. 3: 273–298.

Vassiliadis, P. 1998. "Modeling Multidimensional Databases, Cubes, and Cube Operations." In *Proceedings of the Tenth International Conference on Scientific and Statistical Database Management*, ed. M. Rafanelli and M. Jarke, 53–62. Los Alamitos, CA: IEEE Computer Society.

Vetterli, T., A. Vaduva, and M. Staudt. 2000. "Meta Data Standards for Data Warehousing: Open Information Model vs. Common Warehouse Metamodel." *SIGMOD Record* 29, no. 3: 68–75.

Wiederhold, G. 1992. "Mediators in the Architecture of Future Information Systems." *IEEE Computer* 25, no. 3: 38–49.

9 A Conceptual Information Model for the Chemical Process Design Lifecycle

Birgit Bayer and Wolfgang Marquardt

9.1 Introduction

Demand is growing in process industries for an improvement of process design that results in both shorter development cycles and better plants. The current state of the art in design support is based mainly on separate or loosely linked software packages, which are applied for special tasks and purposes. With these software tools, growing amounts of data, documents, and all other kinds of information are handled. Flow charts, process descriptions, equipment specifications, experimental data, mathematical models, simulation results, cost calculations, safety reports, and documentation are created, used, and stored in different and often very specialized software systems in proprietary formats. These different pieces of information need to be managed, since they are valuable resources of knowledge. Although they are created and handled within different tools, there are often dependencies and redundancies among these pieces of information. Thus, automated information exchange has been recognized as being of major importance for improving and enhancing engineering work (Beßling et al. 1997). Empirical studies have shown that companies are working toward integrated solutions for the management of information, but that there are still unsolved problems. These include the data exchange between heterogeneous tools and the integration of different life cycle phases (Hameri and Nihtilä 1998).

For the integration of existing software tools and the development of software environments that provide central services and support functionalities, a thorough understanding of the domain to which the software will be applied is necessary. The tools, the information handled within these tools, and the work processes using that information need to be understood together with their dependencies. Such an understanding can be acquired best by the development of a conceptual information model.

Information modeling is a commonly used method for the analysis and formalization of information structures as the basis for software design (Mylopoulos 1998).

An information model for the chemical process design life cycle has to be powerful enough to represent all data created and used during the design process. As a high-quality data model, it should be correct, minimal, and understandable. Since the domain of chemical process design—like any other engineering domain—contains a huge amount of information, a model designed for its representation must be designed for extensibility (McKay, Bloor, and de Pennington 1996). Therefore, we propose the development of a metamodel, in addition to the model itself. Within this metamodel, which represents the model of the particular information model being considered, semantical details are abstracted. Types of classes, attributes, and methods can be defined for later use in information models. Also, recurring structural patterns and symmetries can be formalized. The metamodel provides a structure and an organizational scheme for the whole model and its data instances. The class level of the information model itself must provide a rich semantic description of individual concepts in order to provide design knowledge for the tools that support the design process. From our point of view, an extensible data model must aim at describing the universe of discourse precisely in the sense of an ontology (Uschold and Gruninger 1996). It must abstract from specific purposes. Numerous approaches to data models for chemical engineering and related disciplines have been presented in the past (see Bayer and Marquardt 2003 for a critical review). Each of the data models in these approaches has a specific and original scope and is built for a distinct purpose. Thus, its coverage is limited compared to the information that is handled during the design life cycle of chemical plants.

A conceptual data model for the products, the documents, and the work processes of the design life cycle of chemical processes has been developed. This model, called CLiP (Conceptual Lifecycle Process), has been designed within the Collaborative Research Center IMPROVE (Information Technology Support for Collaborative and Distributed Design Processes in Chemical Engineering) (Marquardt and Nagl 1998; Nagl and Marquardt 2001) as a basis for the understanding of design processes in chemical engineering and for the development of specific computer-based support tools. CLiP is independent of any specific implementation within a software system. It forms a model framework that can be used as the basis for the integration of existing information models and for future modeling activities (Bayer and Marquardt 2004). Therefore, the focus is set on the description of general concepts within the domain of chemical engineering and their dependencies.

This chapter provides an overview of CLiP. In the following section, the model framework itself is presented, along with its main concepts and their relations. This framework has been modeled in O-Telos using the ConceptBase system (Jeusfeld et al. 1998).

It is then shown how the metamodel provides a means of structuring a detailed information model for the chemical process design life cycle. Not only has the informa-

tion created and used during design processes been considered within CLiP, but also the design processes themselves. Therefore, concepts to describe activities and documents handled within activities have been introduced. They are presented after the model framework itself. The last section of the chapter introduces some more detailed concepts that have been derived to represent knowledge about chemical processes as a basis for design support functionalities. In that section, some advantages and drawbacks of ConceptBase are discussed that led to the decision to model parts of CLiP within ConceptBase and others with UML (Rumbaugh, Jacobson, and Booch 1999).

9.2 The Model Framework CLiP

sources

The framework of CLiP covers three metamodel levels on which modeling concepts are given at different degrees of abstraction: the *meta-metalevel*, holding the general system; the *metalevel*, with technical, material, and social systems; and the *simple class level*, with the chemical process system. Metadata are used to understand and describe data and the use of data in information systems (Jarke et al. 2000). There are different possible specifications of metadata; here, metamodels are specified in the sense of a dictionary to describe data elements and relations among them. The two metalevels of CLiP have been developed independently from the domain of chemical engineering. They represent ideas of systems theory and systems engineering (e.g. Bunge 1979; Patzak 1982).

9.2.1 Overview

`System` is introduced on the meta-metalevel as the root concept of the model framework (see figure 9.1). Different kinds of systems can be distinguished on the metalevel. `TechnicalSystems` represent all kinds of technical artifacts that are built to fulfill some functionality. Technical systems are either `Devices` or `Connections`. Devices hold the major functionality and are linked by connections. Furthermore, `MaterialSystem` and `SocialSystem` are introduced as instances of `System`. A material system abstracts matter and substances, which can be used in various manners by technical systems. A `SocialSystem` can be a group of persons or a single person.

On the simple class level, the concept of `TechnicalSystem` is further refined to `ChemicalProcessSystems`, which consist of three distinguished parts: the `ProcessingSubsystem`, `OperatingSubsystem`, and `ManagementSystem`. There are two different instantiations of `MaterialSystem` at this level of detail: `ProcessingMaterial`, which is processed in order to get a specified product, and `ConstructionMaterial` (not shown in figure 9.1), used to build the `ChemicalProcessSystem`. The behavior of processing material can be described by `MaterialModels`. These can, for example, be referenced by `ProcessingSubsystemModels`

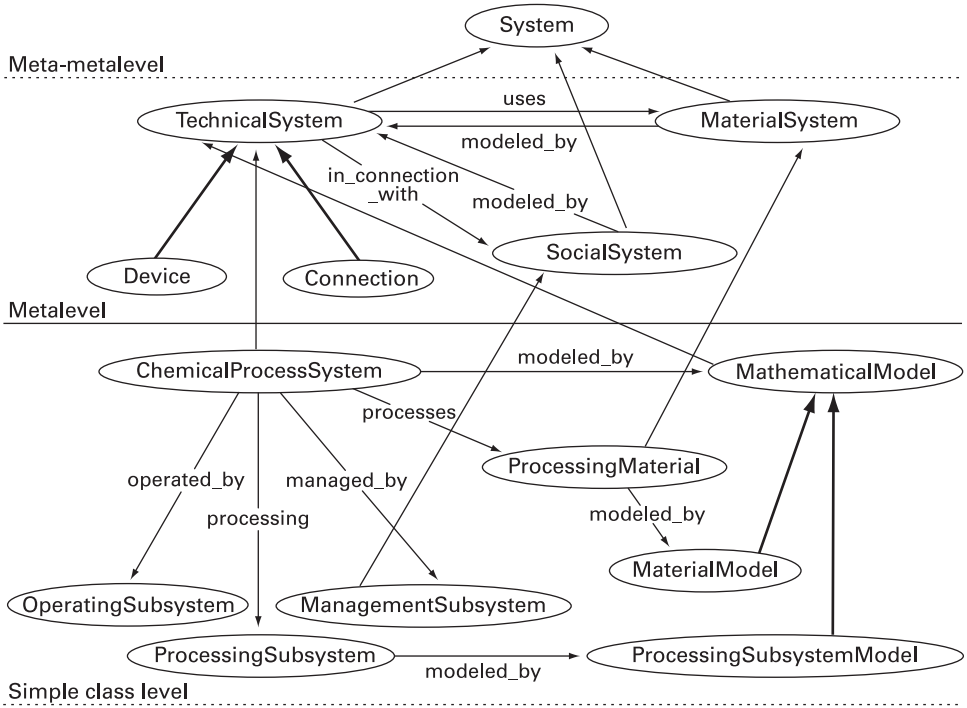


Figure 9.1
The three metalevels of CLiP

describing the `ProcessingSubsystem`. Material models and processing subsystem models are examples of `MathematicalModels`, which are instances of `Technical-System`.

9.2.2 The General System

The root concept of CLiP, from which all other modeling concepts can be derived, is the system. There are numerous definitions of systems given in the literature in the area of systems theory and systems engineering. They all share some common concepts that can be summarized by the following definition, according to Patzak (1982, 19) translated from German by the authors: "A system consists of a set of elements which have properties and which are concatenated through relations." This definition is the basis for the model of any system in CLiP (figure 9.2). The system contains elements that are systems themselves. Systems refer to other systems. These associations refer to the fact that systems and their elements can interact with or can be related to other systems, which are either part of the same system or belong to the systems environment (Bunge 1979). A special kind of relationship between systems is

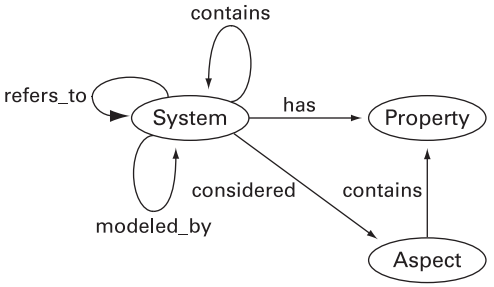


Figure 9.2
The general system (meta-metalevel)

the `modeled_by` association. A system can be a model of another system. This means that it can be used to describe and predict the properties of the original system with several (known and unknown) simplifications and assumptions (see, for example, Minsky 1965). This very abstract introduction of the model as a system includes all kinds of models, like mathematical models, real systems that are a copy of another system at a different scale, and even information models.

Every system has at least one `Property`. In different contexts during a system's life cycle, different properties are of interest. These can be grouped together under an `Aspect`. Thus, different aspects of a system, which contain one or more different properties, as shown in figure 9.2, can be considered.

9.2.3 Technical Systems

Three different kinds of systems are introduced on the metalevel of the modeling framework (see figure 9.1): the technical system, the material system, and the social system. The technical system is an artifact that is designed to fulfill some required function. Examples of technical systems are chemical plants, cars, computer systems, infrastructure systems like sewage systems, and mathematical models.

The function of a technical system is its ability to transform some input into a required output; it is specified during system design. Technical systems (see figure 9.3) can contain other technical systems (for example, a sewage system contains a purification plant), and they can interact with one another (e.g., the interaction among single computers in a network). A technical system can be a model of another technical system; examples are a mathematical model of the power train of a car used to predict the car's acceleration times and final speed and a pilot plant as a physical model of an industrial-scale plant.

Within a technical system, two classes of subsystems can be distinguished: Devices, which hold the major functionality, and connections, which link the different devices together. Both devices and connections can be decomposed. A device can contain

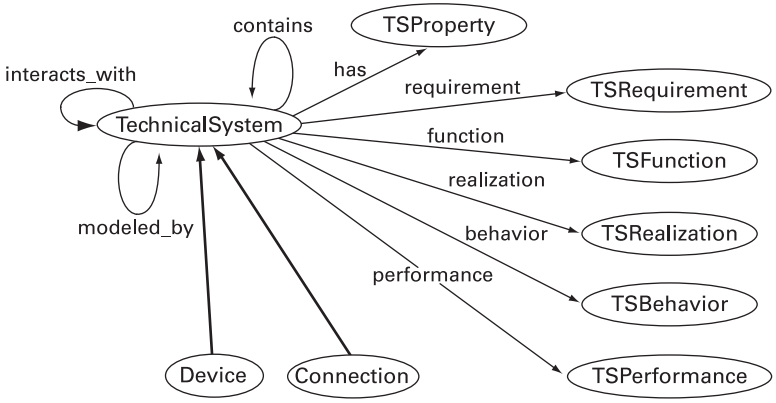


Figure 9.3
The technical system (metalevel)

one or more devices and some connections. When two or more devices are combined into one, at least one connection is needed to connect the devices. A connection can also be formed from the combination of two or more connections; there is always a device between each pair of connections.

The properties of a technical system are named `TSProperty`, with `TS` standing for "technical system." Five aspects of technical system properties that are of major importance during the design life cycle can be distinguished: `TSRequirement`, `TSFunction`, `TSRealization`, `TSBehavior`, and `TSPerformance`. Technical system requirements describe the desired function and behavior of the system being designed and how the system should be constituted. Technical system function refers to the desired or planned function of the technical system (the actual function of the system during its use and operation is described by technical system performance). A technical system realization is the development of the system in a specified manner so that all functions of the technical system are fulfilled. It covers geometrical, mechanical, physical and organizational properties (Patzak 1982). Technical system behavior describes how the system behaves under certain circumstances. The behavior results from the planned function and the realization of the technical system as well as from its environment. Technical system performance refers to the analysis and description of the technical system with respect to the requirements. Here, the behavior is abstracted under a specific view in order to determine whether the realized system performs the functions given in the requirements and if the design process was successful. The costs that can be associated with a technical system, its design, and its usage (`TSCosts`) are a special type of performance; they are of importance during the design and use of a technical system, since economic evaluations strongly influence design decisions.

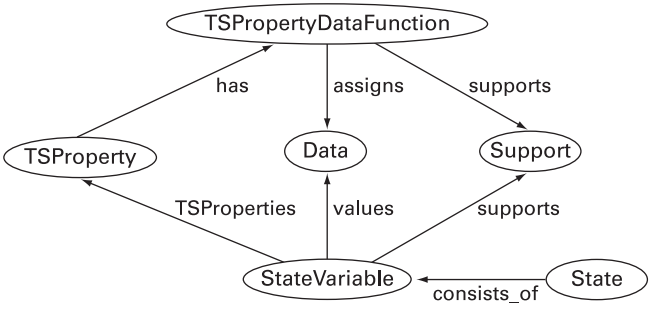


Figure 9.4
Technical system properties and data

Every property has multiple possible appearances given as `Data` (see figure 9.4). At any specific observation, each property can be described only by specific data. For the identification of that observation, the concept of `Support` has been introduced, by means of which the background of an observation can be given (Klir 1985). Most often, the background is a specific time or a spatial coordinate characterizing an observation. The complex relations among technical system property, data, and support are described explicitly by the technical system property data function. This property data function assigns data from a set of possible data actually observed at a specific support to a property. It can refer, in the case of measurements, to a data log in which the data of a property are recorded for different supports, usually over some time interval. For numerical simulations, the property data function can refer to a mathematical function.

In order to illustrate the complex information model given in figure 9.4, we use the small example of a traffic light. One of the properties of this technical system is the mode of the upper light. Possible data values of this property are "on" and "off." One function is to turn the traffic light to red, that is, to change the data of the upper light from "off" to "on" at a distinct time. This function does not change the property; there is still the mode of the upper light as a property of the system. Also, the possible data values are not changed; they are still "on" and "off." Rather, the relation between the property and the data is changed between two different points in time (i.e., for two different supports). A technical system property together with its data at a distinct point in time can be defined as a `StateVariable` of the technical system. The totality of all state variables builds the `State` of the technical system. Thus, the state specifies a technical system at one point in time. The change of the state with time describes the system's behavior.

The function of a technical system can change the `TSPropertyDataFunction`; that is, the data and support of some properties of the technical system are changed

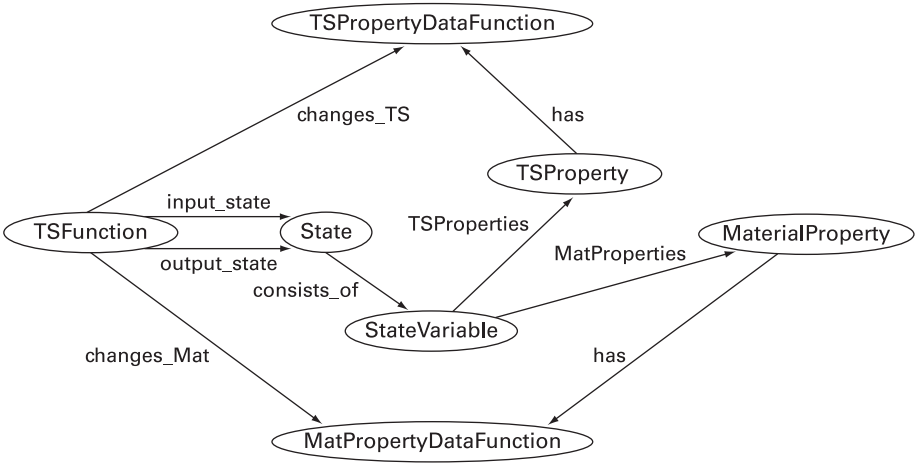


Figure 9.5
Input and output of technical system functions (metalevel)

(figure 9.5). Not only the properties and states of a technical system are influenced by its functions, but also those of material, energy, or information processed within the technical system (Koller 1975). In the area of chemical engineering, the transformation of material is the most important function. Therefore, it is explicitly modeled that a technical system function can change the `TSPropertyDataFunction` as well as a material property data function (`MatPropertyDataFunction`), which relates a `MaterialProperty` to data. The material properties together with their data and the support are relevant state variables in chemical engineering, in addition to the technical system properties themselves. Thus, the state of a technical system is composed not only of state variables derived from technical system properties but also from those of material properties (see figure 9.5). An input state and an output state can be defined for a technical system function holding the properties, data, and support related to another via the property data functions before and after their change through the function.

9.2.4 Functions of Technical Systems

During the design of a technical system, the detailed specification of the different functions that need to be fulfilled by that system is a very important and complex task. The specification of the functions and their association with the individual parts of the system's realization influence to a large degree the behavior and thus the performance of the system. Therefore, the categorization of functions is an important step toward the representation of knowledge for technical design processes like chemical process design.

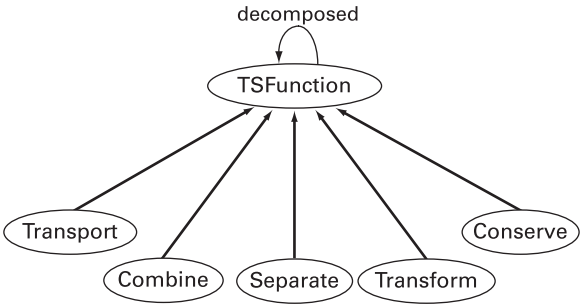


Figure 9.6
Basic functions of technical systems (metalevel)

In different technical domains, classifications of basic and elementary functions are given (see, for example, Blair and Whitston 1971; Koller 1975). Patzak (1982) combines these classifications into a general one in which basic functions are given that are independent of the domain to which the technical system belongs. In figure 9.6, these domain-independent classes of technical system functions are given: `Transport`, a change in position and time; `Combine` and `Separate`, changes in amount and composition, respectively; `Transform` a change in form, kind, and character; and finally `Conserve`, a function applied to a system with the intention that its state not be changed. The `TSFunctions` shown in figure 9.6 can be applied to material systems, energy systems, or information systems.

9.2.5 Chemical Process Systems

On the simple class level, the concept of technical systems is further refined to the chemical process system, which consists of three distinguished parts (Backx, Bosgra, and Marquardt 1998): `ProcessingSubsystem`, `OperatingSubsystem`, and `ManagementSystem` (figure 9.7). The processing subsystem holds functionalities of materials processing; the operating subsystem comprises the technology for controlling this processing subsystem; and finally, the management system refers to the `Personnel` working on the chemical plant. `ProcessingSubsystem` and `OperatingSubsystem` are instances of `TechnicalSystem`, whereas `ManagementSystem` is an instance of `SocialSystem`. The function of the chemical process system is given by `ChemicalProcessElements`. Further aspects of the chemical process system are `ChemicalProcessRealization`, `ChemicalProcessBehavior`, and `ChemicalProcessPerformance` with `ChemicalProcessSystemCosts`. In figure 9.8, the chemical process system is given with its subsystems and associated systems and their functions and realizations; the figure shows how these contribute to the overall function (the chemical process elements) and the chemical process realization. The chemical process elements can partially be mapped onto `ProcessSteps` that are the

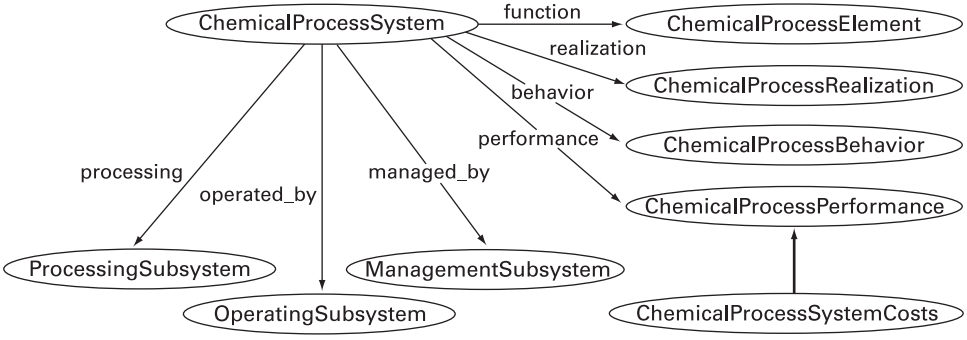


Figure 9.7
Chemical process system with its subsystems and properties (simple class level)

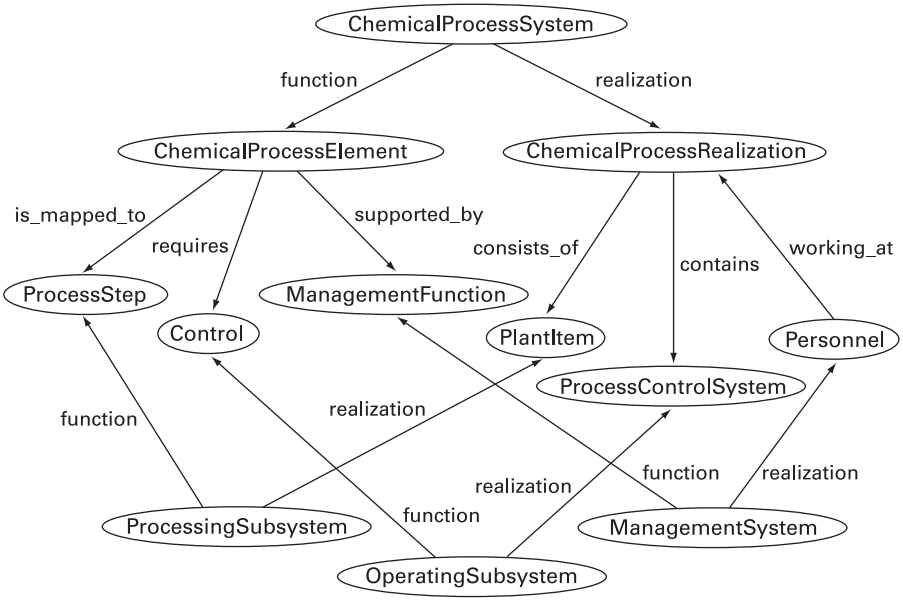


Figure 9.8
Chemical process system with its functions and realizations (simple class level)

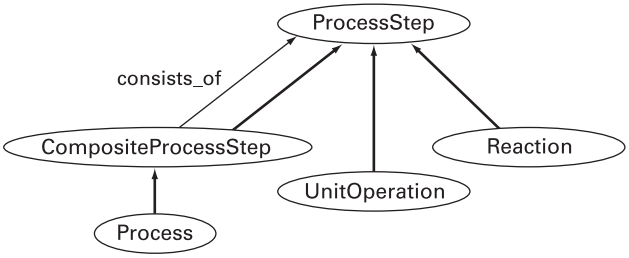


Figure 9.9
Process steps (simple class level)

functions of the processing subsystem. Such a mapping can be done for most chemical process elements, since the major function of the chemical process system is identical to the function of the processing subsystem (i.e., the processing of some material in order to obtain a specified chemical product); the functions of the operating subsystem, the `Control`, and the `ManagementFunctions` are further functions that are required in order to obtain the overall function.

The chemical process realization consists of the plant built from numerous `PlantItems` and contains the `ProcessControlSystem` (the realizations of `ProcessingSubsystem` and `OperatingSubsystem`, respectively). `Personnel` (the "realization" of the management system) are working toward the realization of this chemical process.

9.2.6 Functions of Chemical Process Systems

Process steps represent the functions of the processing subsystem (see figure 9.8). Within chemical process systems, material is processed in order to produce some specified compound from raw materials. This processing comprises physical, chemical, and biological procedures that are performed in a specific order. These procedures can be subsumed as `ProcessSteps` (figure 9.9). Separation, agglomeration, and absorption are examples of process steps. Chemical and biological procedures are described by `Reactions`. `UnitOperations` are elementary process steps based on physical phenomena. Unit operations, reactions, and other process steps can be combined into complex procedures represented by `CompositeProcessSteps` and complete chemical `Processes`.

Although the number of chemical processes is enormous, there is a relatively small number of unit operations that can be combined into any kind of process. Several chemical engineering handbooks deal with unit operations and their realization in particular equipment (e.g., Perry and Chilton 1984). The categorizations of unit operations they give can be used accordingly to develop a class hierarchy. The data model developed by the Process Data Exchange Institute (pdXi) initiative (ISO

1998) also includes a class hierarchy of unit operations. Technische Güte und Lieferbedingungen (TGL) 25000 (TGL 1974) was a standard of the German Democratic Republic that gave an exhaustive description of unit operations, their applications, and the equipment required for their implementation. Two classifications of unit operations were given in that standard: classification schema A, in which the physical state of the processed processing materials serves as the organizing principle, and classification schema B, in which the unit operations are organized according to theoretical chemical engineering considerations. Both classification schemas have been integrated into CLiP with the goal of describing knowledge about chemical processes and making it available in a database to support design processes (e.g., for selecting appropriate process steps for a given task within a chemical process). This integration is discussed in detail later in the chapter.

9.2.7 Partial Model Structure of CLiP

On the simple class level, CLiP is divided into partial models holding concepts that belong together in the sense that they describe one distinct part of the domain of interest completely. Figure 9.10 shows the partial models related to the chemical process system; these partial models correspond to some of the concepts introduced in figure 9.1: The main concepts of CLiP introduced on the different metalevels provide the overall structure of the model. Within the partial models of the chemical process system and its three (sub)systems, all information about these systems is given. This information comprises the different properties of the systems. The concepts related to these properties are again grouped into partial models that are nested within the partial model of the system itself. In figure 9.10, three partial models are shown that are used to model different properties of the chemical process system: function, realization, and behavior.

The partial models can be set up and used largely independently of one another. Still, there are a lot of interdependencies among them. These interdependencies are modeled explicitly in CLiP with associations between concepts belonging to different partial models, thus defining the interfaces between the partial models. By dividing the model into parts and reintegrating these with associations, an open and extensible model structure is obtained. New partial models can be introduced by developing them independently and then describing their relations to the existing ones. Thus, partial models provide additional structuring functionality in the metamodeling framework.

9.3 Workflow Modeling within CLiP

In addition to the concepts describing technical systems with their properties in detail, CLiP also covers modeling concepts that allow the representation of the flows

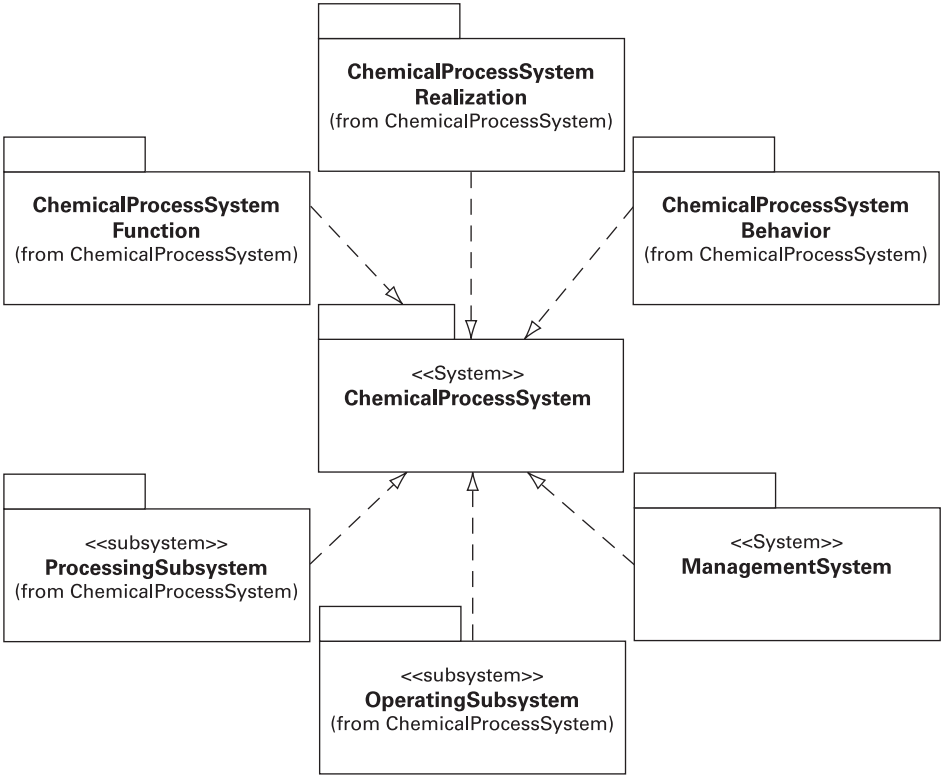


Figure 9.10
Partial models at the simple class level of CLiP (represented by UML packages)

of the work performed by designers during chemical process design and of the information that is processed within these workflows. Two concepts, namely, documents and activities, have been introduced for this purpose. They are presented in this section, together with their relations to the information models discussed so far.

9.3.1 Documents

Within the metamodel level of CLiP, concepts have been introduced to describe documents and their relations to information about different systems (see figure 9.11). A `Document` can contain numerous technical system properties (instances of `TSProperty`), social system properties (instances of `SocSysProperty`), and material properties (instances of `MaterialProperty`), together with their data and the support that is needed to distinguish different observations of these properties.

Within the chemical process design life cycle, activities are steps in a work process that create, modify, or delete some kind of input information and thereby produce an

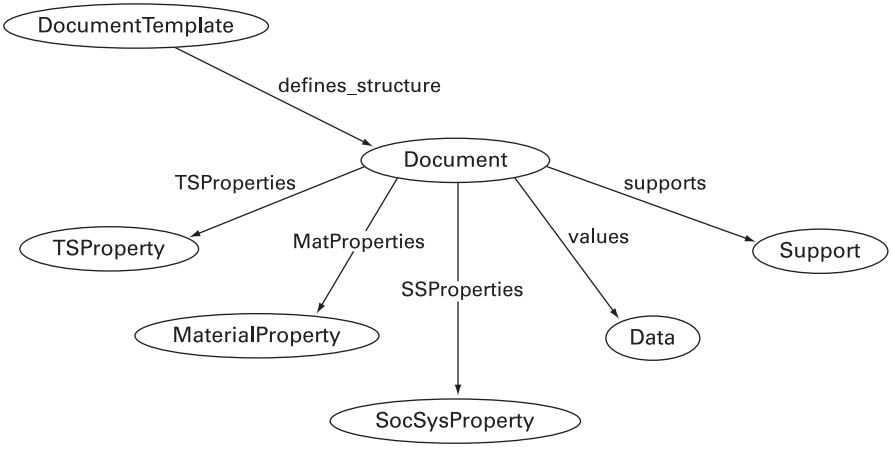


Figure 9.11
Documents within CLiP (meta-metalevel)

output. They represent manipulations of information by an engineer or some software. This is described within CLiP by assigning input and output documents to a particular activity. These documents hold the properties of the system on which the activity operates. The content of these documents is read and interpreted and may be changed during the execution of the activity.

Many documents that are used in technical design processes have a (partially) predefined content and format. The format is often specified, either through the software tool that is working on the document or by some standard. Therefore, `DocumentTemplates` can be defined for many documents. Document templates refer to the properties and the support that must be specified for a particular document (not shown in figure 9.11) but do not contain any data for the properties. Thus, the structure of a document can be defined in a document template.

9.3.2 Activities

A social system is introduced on the metalevel as one special type of system. It represents groups of persons (teams, societies) or individuals. The most important aspect of social systems in design processes is the `Activity`, which is one of their functions (see figure 9.12). An activity is performed in order to reach a given `Goal`. It can be decomposed into subactivities. A temporal or logical order can be given by prerequisites and through scheduling conditions (not shown in the figure). The execution of an activity requires resources; these are mainly technical systems like computers, software programs, or experimental setups. An activity is executed by an `Actor`, either an individual person or a team, who may have to possess `Skills` in order to be able

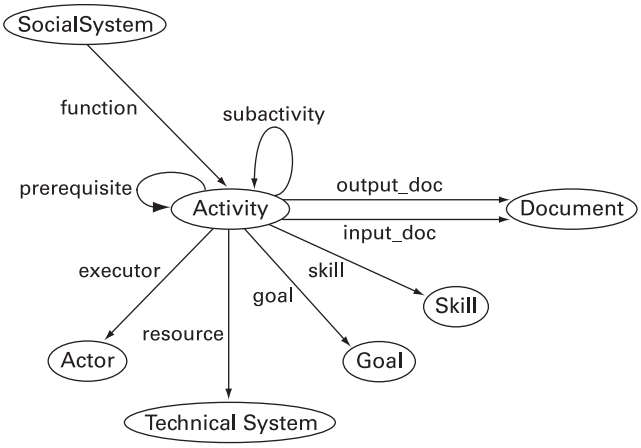


Figure 9.12
Social systems and activities in the context of technical system design (metalevel)

to perform the execution. A more detailed description of activities and their characteristics is given by Eggersmann, Krobb, and Marquardt (2000).

Three basic types of activities performed during a design process (not shown in figure 9.12) can be distinguished: `Synthesis`, a creative and constructive activity in which a system or some data of a system property are created or set; `Analysis`, a descriptive activity in which the system and its properties with their data are investigated and screened; and `Decision`, the judgment about the system based on the results of the analysis (Eggersmann et al. 2001).

The relations of activities that are performed during the use of a technical system (e.g., during the operation of a chemical plant) to the technical system properties are different from those for design activities. During a system's use, activities are not performed to create and modify documents that describe the system. Rather, interventions into the technical system aim at influencing and changing its function and behavior. Such activities modify the state of the system. This is modeled within CLiP by means of a change of the technical system property data function affected by an activity (see figure 9.13). The activity model shown in figures 9.12 and 9.13 allows the definition of workflow models, in which the different activities performed (e.g., during process design) can be given in a specified order together with the input and output information they are working on and the technical system that is influenced by the execution of each activity. Since activities can be decomposed into subactivities, the description of work processes at different levels of detail is possible, ranging from enterprise models and workflow models to detailed activity models in which the single activities of an actor are described. Since activities do not have to

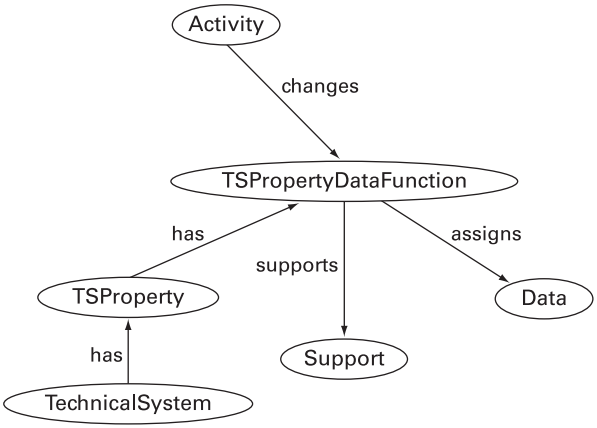


Figure 9.13
Activities during use of a technical system (metalevel)

be connected by control flows, the description of ill-defined work processes is also possible. Here, the available input documents reflect a certain contextual situation and provide information as to whether a particular activity is executable or not. Thus, CLiP provides flexible possibilities for the modeling of workflows and activities as a basis for the understanding and analysis of different work processes. CLiP models can then be used to develop different support functionalities ranging from administrative workflow support to tracing and support of fine-grained work processes. Also, operating procedures for chemical plants can be described, and the description can be used to support and automate process operation and control, resulting in dynamic and flexible production processes.

9.4 Representation of Domain Knowledge Using ConceptBase

In addition to the description of a domain by means of the introduction of major concepts and the depiction of their dependencies, information models can also be used to represent domain knowledge. Within ConceptBase, this can be accomplished through the introduction of constraints and rules, which give more detailed information about individual concepts and their dependencies. Also, ConceptBase's consistent support of multiple inheritance and instantiation can be used to implement powerful class hierarchies. The following subsections show such a representation of domain knowledge using the example of process steps according to the aforementioned TGL 25000 (TGL 1974). Furthermore, some drawbacks of ConceptBase are discussed that led to the decision to model the detailed class diagrams of CLiP with UML rather than ConceptBase.

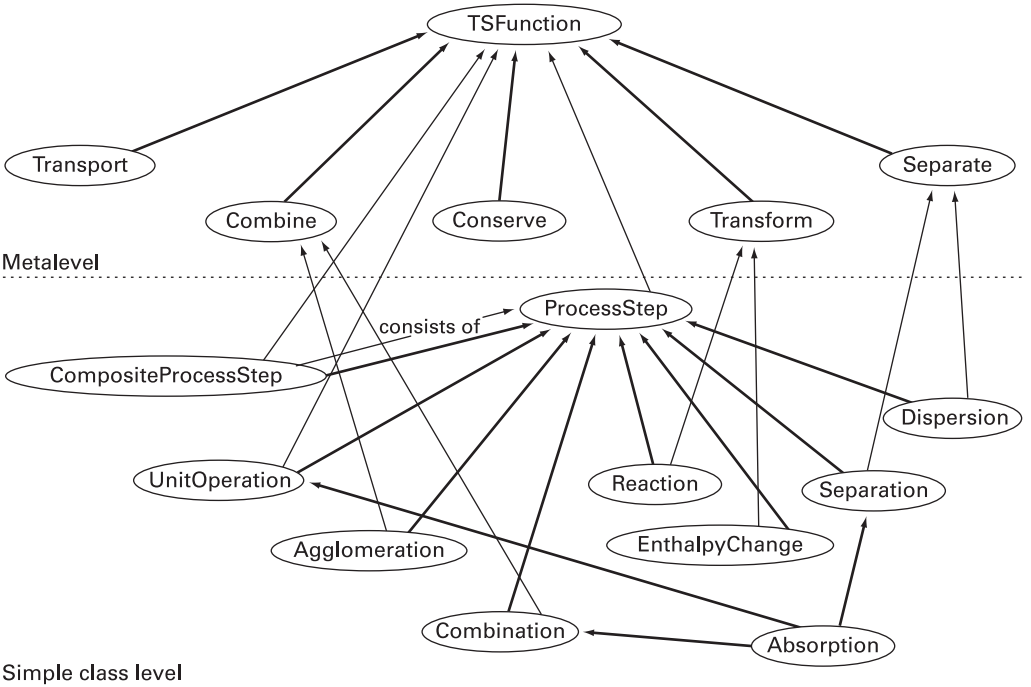


Figure 9.14
Classification schema A of TGL 25000

9.4.1 Classification of Chemical Process Functions

As noted in section 9.2.6, both schemas A and B for classifying unit operations of TGL 25000 were integrated within CLiP and implemented with ConceptBase. The goal of this modeling effort was the description of knowledge about chemical processes to make it available in a database for supporting design processes (e.g., for the selection of appropriate process steps for a given task within a chemical process).

Figure 9.14 provides an overview of the implementation of classification schema A. TGL 25000 distinguishes five major types of unit operations: `EnthalpyChange`, `Combination`, `Agglomeration`, `Separation`, and `Dispersion`. Since these types of functions used within chemical processes can be composite, they are introduced within CLiP as subclasses of `ProcessStep` and not of `UnitOperation`, since the latter is defined to be elementary. The different unit operations given in TGL 25000 are subclasses of `UnitOperation` and of the type of `ProcessStep` to which they belong (see, for example, the unit operation `Absorption` in figure 9.14).

Classification schema B of TGL 25000, in which the unit operations are classified according to the active principle and phenomena on which they are based, is

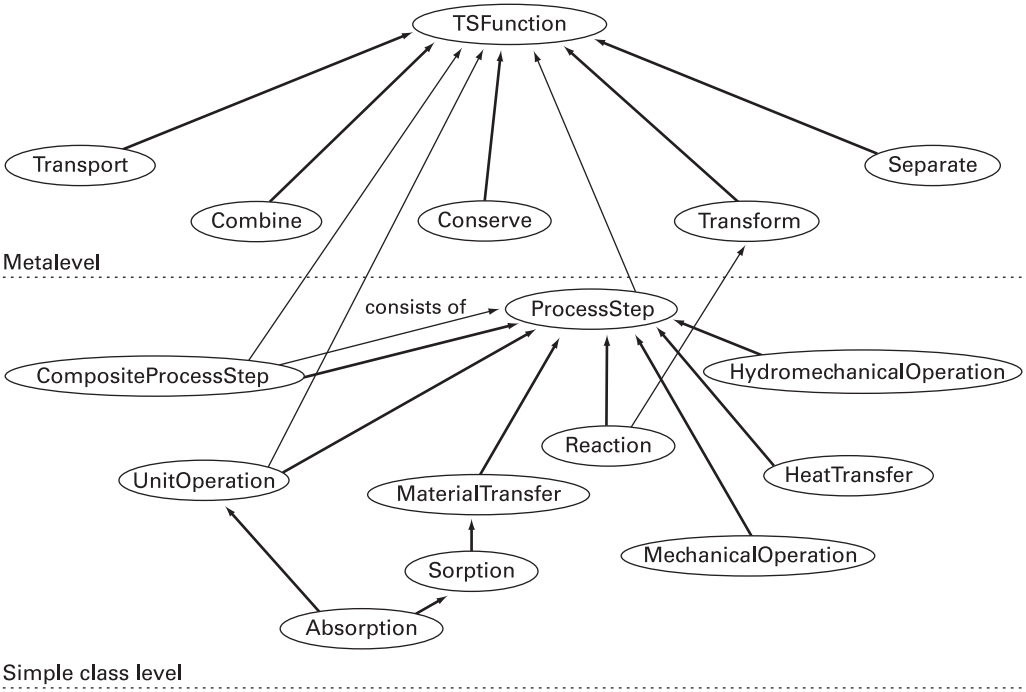


Figure 9.15
Classification schema B of TGL 25000

sketched in figure 9.15. Within this schema, the four basic types of procedures that are used to classify the unit operations are `HydromechanicalOperation`, `HeatTransfer`, `MaterialTransfer`, and `MechanicalOperation`. Since these types of procedures can be composite and complex, they are introduced as subclasses of `ProcessStep` and not of `UnitOperation`. Different subclasses of the four major types of procedures are given in classification schema B that further specify the nature of the procedures and the phenomena on which they are based on; one example is `Sorption`, which is a subclass of `MaterialTransfer`. The different unit operations given in TGL 25000 are subclasses of `UnitOperation` and of the type of `ProcessStep` to which they belong (see, for example, `Absorption` in figure 9.15).

9.4.2 Multiple Inheritance and Instantiation

Since the classification principles of the two schemas given in TGL 25000 are different, the subclasses of process steps according to classification schema B are instantiations of different types of `TSFunctions` than the ones according to classification schema A (compare figures 9.14 and 9.15). The single unit operations inherit these instantiations links.

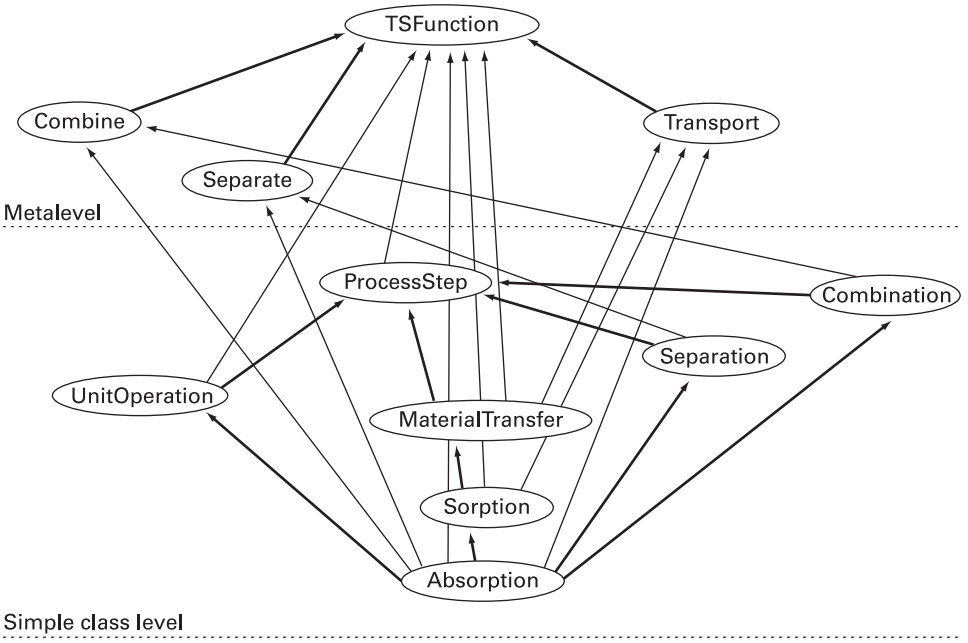


Figure 9.16
`Absorption`, with its classes and superclasses

Within ConceptBase, both classification schemas of TGL 25000 have been implemented as presented here. Because of the two different specialization principles, this led to a very complex data model with numerous multiple instantiations and multiple specializations. An example is shown in figure 9.16: `Absorption` is a `UnitOperation` that is both a `Separation` and a `Combination` according to classification schema A. Following classification schema B, `Absorption` is a `Sorption` that is a `MaterialTransfer`. Thus, `Absorption` has four superclasses. Since `Separation` is an instance of `Separate`, `Combination` is an instance of `Combine`, and `Sorption` is an instance of `Transport`, `Absorption` is an instance of these three metaclasses (and of course of `TSFunction` as well).

9.4.3 Rules and Constraints

Figures 9.14 and 9.15 show the major concepts of classification schemas A and B of TGL 25000. Within both schemas, further subclasses of unit operations are introduced and classified according to the materials processed or the phenomena that occur. Within classification schema A, these classifications could be modeled by introducing subclasses like `SeparationOfLiquids` and `SeparationOfLiquidsToLiquidAndGas`. This would lead to a large number of abstract classes and a complex

hierarchy with numerous multiple inheritances. Advanced modeling functionalities of ConceptBase like constraints and rules can be introduced for each specific unit operation limiting the application of the operation to a particular processing material at a specific aggregate state and thus reducing the need for multiple inheritance.

The following frame definition of the unit operation `Absorption` together with its constraints illustrates this (for an introduction to the O-Telos language used in ConceptBase, see Jarke, Jeusfeld, and Quix 1998):

```
SimpleClass Combination in Class,Combine isA ProcessStep with
  attribute
    combined_to: AggregateState;
    combined_from1: AggregateState;
    combined_from2: AggregateState
end

SimpleClass Separation in Class,Separate isA ProcessStep with
  attribute
    separated_from: AggregateState;
    separated_to1: AggregateState;
    separated_to2: AggregateState
end

SimpleClass Absorption in Class,Separate,Combine isA
Separation,Combination with
  constraint
    applicability_sep:
    $ forall u/Absorption a/AggregateState
    (u separated_from a)
    ==> (a == Gas) $;
    separated_phases:
    $ forall u/Absorption a,b,c/AggregateState
    (u separated_from a) and (u separated_tol b) and
    (u separated_to2 c)
    ==> ( ( (a == Gas) and (b == Liquid) and (c == Gas) ) or
    ( (a == Gas) and (b == Gas) and (c == Liquid) ) ) $;
    applicability_comb:
    $ forall u/Absorption a/AggregateState
    (u combined_to a)
    ==> (a == Liquid) $;
    combined_phases:
    $ forall u/Absorption a,b,c/AggregateState
```

```
    (u combined_to a) and (u combined_from1 b) and
    (u combined_from2 c)
    ==> ( ( (a == Liquid) and (b == Liquid) and (c == Gas) ) or
    ( (a == Liquid) and (b == Gas) and (c == Liquid) ) ) $
end
```

Absorption consists of the transfer of some chemical component from a gas phase into a liquid phase to purify the gas phase. It thus can be interpreted as a separation from gas to liquid and gas as well as a combination of a liquid and a gas phase into a liquid phase. Therefore, `Absorption` is a specialization of `Separation` and of `Combination` (and is thus an instance of `Separate` and of `Combine`) inheriting all their attributes, which are also shown in the frame definition.

Four constraints on `Absorption` are defined: `applicability_sep` limits the application of absorption to the separation of gases; `separated_phases` ensures that the gas is separated into a liquid and a gas phase; `applicability_comb` limits the application to combining the inputs into a liquid phase; and `combined_phases` ensures that a liquid and a gas are combined into a liquid. These four constraints ensure that all instantiations of `Absorption` are in accordance with the definition of absorption given in TGL 25000. Similar constraints can be defined for all other unit operations. This kind of implementation of domain knowledge within a database for chemical process design guarantees that every process step created fulfills a required function.

An alternative to the introduction of constraints for implementing the knowledge given in TGL 25000 into ConceptBase is the introduction of rules that define which unit operations can be applied to which process steps. To illustrate this, we examine the three unit operations absorption, gas centrifugation, and decantation, which are all special types of separations. Within the class `Separation`, for each of these unit operations a rule is defined specifying which kind of separation they can implement:

```
SimpleClass Absorption in Class,TSFunction isA UnitOperation end

SimpleClass GasCentrifugation in Class,TSFunction isA
UnitOperation end

SimpleClass Decantation in Class,TSFunction isA UnitOperation end

SimpleClass Separation in Class,Separate isA ProcessStep with
  attribute
    separated_from: AggregateState;
    separated_to1: AggregateState;
    separated_to2: AggregateState;
    applicableUO: UnitOperation
```

```
  rule
    AbsorptionRule:
    $ forall s/Separation
    (s separated_from Gas)
    ==> (s applicableUO Absorption) $;
    GascentrifugationRule:
    $ forall s/Separation
    (s separated_from Gas)
    ==> (s applicableUO GasCentrifugation) $
    DecantationRule:
    $ forall s/Separation
    (s separated_from Liquid)
    ==> (s applicableUO Decantation) $;
end

Token SomeSeparation in Class,Separation with
  separated_from
    sep_from: Gas
end

QueryClass SeparateGas isA UnitOperation with
  constraint
    function:
    $ exists t/Separation
    (t separated_from Gas) and (t applicableUO this) $
end
```

The absorption rule says that absorption is applicable to the separation of a gas; the gas centrifugation rule says the same for gas centrifugation. The decantation rule states that a decantation can be used to separate a liquid. If these rules are implemented within a knowledge base, queries can be processed, for example, to get a list of all possible unit operations that can be applied to the task of separation of a gaseous input (in this example, the result of such a query will be absorption and gas centrifugation, but not decantation). So this implementation can be used as a basis for design support: The designer can formulate the required process function as a query and will get as the result all process steps that can be applied to fulfill that function in principle.

9.4.4 Discussion of Multiple Inheritance, Rules and Constraints

From a data-modeling point of view, multiple instantiations and multiple specializations are disadvantageous. For example, they can lead to inconsistencies among

attributes inherited from different superclasses having the same name but different values. In such cases, rules and constraints are needed for resolving conflicts (within ConceptBase/Telos, a multiple-generalization/multiple-instantiation axiom is set [Jarke, Jeusfeld, and Quix 1998]). Furthermore, conceptual as well as implementation simplicity is lost with multiple specialization. On the other hand, multiple specialization allows more powerful specifications of classes and thus provides more possibilities for the reuse of information (Rumbaugh et al. 1991).

One has to decide, therefore, how complex the model one is creating may become and in how much detail the knowledge should be represented. In order to ensure consistency of the model according to TGL 25000, constraints like the ones given in section 9.4.3 need to be defined. If the data model will be used as a knowledge base for design, using multiple specialization might be called for, while in cases where the use of the data model will be more lightweight a single classification schema might prove sufficient.

9.5 Conclusions

This chapter has presented the conceptual information model CLiP, covering product data, documents, and activities of chemical engineering design processes. This information model is primarily intended for the analysis of the chemical process design life cycle, ranging from the design of the chemical process itself to the design of the control system and of operating procedures. With their very general structure and abstract level of detail, the metalevels of CLiP can serve as a modeling framework for the development of more detailed data models needed for a specific purpose or even for the integration of existing data models. Through the introduction of constraints and rules, the information model can be extended toward a knowledge base for chemical engineering design.

CLiP has been implemented and formalized with metamodels that are modeled in O-Telos. Thus, the conceptual model framework of CLiP is implementable within ConceptBase. The flexibility of introducing several metamodel layers was a prerequisite for the development of a domain-independent model framework. On the simple class level, this framework has been specialized to represent information related to the chemical process design life cycle. To do this, another feature of ConceptBase has been utilized: Knowledge about process steps and their possible classifications has been formalized through the introduction of constraints and rules, which can be used to ensure consistent process designs or for the development of a knowledge base about chemical process functions.

A complete description of all product data and activities occurring in the design life cycle remains as work to be done. Also, the requirements that can be stated about a technical system before the design process starts have not yet been considered.

Furthermore, the behavior of the chemical process system and its three (sub)systems, together with its relations to function and realization, has not yet been covered in sufficient detail. The mappings between the performance of the realized system and the requirements that are stated at the beginning of the design process are of special interest.

Information models like CLiP are a prerequisite for the development of specific application tools and information management facilities and for the integration of tools into software environments. But for these software development tasks, information models on a different level of conceptualization and formalization than the one provided by CLiP are required. More detailed models are needed, for example, to describe the internal data structures of the tools and their functionalities. These have to be consistent with the conceptual information model presented here, which has been developed from the perspective of an application expert in order to describe his domain of interest. Therefore, further model development, refinement, and integration are needed on different levels. These modeling issues are currently investigated within the Collaborative Research Center IMPROVE.

Acknowledgments

We gratefully acknowledge the financial support of the Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center (SFB) 476, ''Information Technology Support For Collaborative and Distributed Design Processes in Chemical Engineering.'' Furthermore, we would like to thank M. Eggersmann, C. Krobb, R. Schneider, and L. von Wedel for many fruitful discussions. Finally, we are very grateful to Jan Morbach for his help with the final editing of the manuscript.

References

Backx, T., O. Bosgra, and W. Marquardt. 1998. ''Towards Intentional Dynamics in Supply Chain Conscious Process Operations.'' In *Proceedings of the Third International Conference on Foundations of Computer-Aided Process Operations*, ed. J. F. Pekny and G. E. Blau. Available at ⟨http://che.www.ecn.purdue.edu/FOCAPO98/⟩.

Bayer, B., and W. Marquardt. 2003. ''A Comparison of Data Models in Chemical Engineering.'' *Concurrent Engineering: Research and Applications* 11, no. 2: 129–138.

Bayer, B., and W. Marquardt. 2004. ''Towards Integrated Information Models for Data and Documents.'' *Computers & Chemical Engineering* 28, no. 8: 1249–1266.

Beßling, B., B. Lohe, H. Schoenmakers, S. Scholl, and H. Staatz. 1997. ''CAPE in Process Design—Potential and Limitations.'' *Computers & Chemical Engineering* 21 (Suppl.): S17–S21.

Blair, R., and W. Whitston. 1971. *Elements of Industrial Systems*. New York: Prentice Hall.

Bunge, M. 1979. *Ontology II: A World of Systems*. Vol. 4 of *Treatise on Basic Philosophy*. Dordrecht: Riedel.

Eggersmann, M., S. Gonnet, G. Henning, C. Krobb, and H. Leone. 2001. ''Modeling of Actors within a Chemical Engineering Work Process Model.'' In *Proceedings of the International CIRP Design Seminar*, ed. T. Kjellberg, 203–208. Paris: CIRP.

Eggersmann, M., C. Krobb, and W. Marquardt. 2000. "A Modeling Language for Design Processes in Chemical Engineering." In *Conceptual Modeling (ER 2000)* (Lecture Notes in Computer Science 1920), ed. A. H. F. Laender, S. W. Liddle, and V. S. Storey, 369–382. Berlin: Springer.

Hameri, A.-P., and J. Nihtilä. 1998. "Product Data Management—Exploratory Study on State-of-the-Art in One-of-a-Kind Industry." *Computers in Industry* 35, no. 3: 195–206.

ISO (International Organization for Standardization). 1998. ISO 10303, Part 231: Process Engineering Data: Process Design and Process Specifications of Major Equipment. Committee draft ISO TC184/SC4/WG3 N740, International Organization for Standardization, Geneva.

Jarke, M., M. A. Jeusfeld, and C. Quix, eds. 1998. *ConceptBase V5.0 User Manual*. RWTH Aachen, Aachen. Available at ⟨http://www-i5.informatik.rwth-aachen.de/CBdoc/userManual-V50/⟩.

Jarke, M., M. A. Jeusfeld, C. Quix, T. Sellis, and P. Vassiliadis. 2000. "Metadata and Data Warehouse Quality." In *Fundamentals of Data Warehouses*, ed. M. Jarke, M. Lenzerini, Y. Vassiliou, and P. Vassiliadis, 123–178. Berlin: Springer.

Jeusfeld, M. A., M. Jarke, H. W. Nissen, and M. Staudt. 1998. "ConceptBase—Managing Conceptual Models about Information Systems." In *Handbook on Architectures of Information Systems*, ed. P. Bernus and G. Schmidt, 265–285. Berlin: Springer.

Klir, G. J. 1985. *Architecture of Systems Problem Solving*. New York: Plenum.

Koller, R. 1975. "Physikalische Grundfunktionen zur Konzeption technischer Systeme." *Industrie-Anzeiger* 97, no. 17: 321–325.

Marquardt, W., and M. Nagl. 1998. "Tool Integration via Interface Standardization?" *DECHEMA Monographie* 135: 95–126.

McKay, A., M. S. Bloor, and A. de Pennington. 1996. "A Framework for Product Data." *IEEE Transactions on Knowledge and Data Engineering* 8, no. 5: 825–838.

Minsky, M. L. 1965. "Matter, Mind, and Models." In *Proceedings of the International Federation of Information Processing Congress*, ed. A. Kalenich, 1: 45–49. Washington, DC: Spartan.

Mylopoulos, J. 1998. "Information Modeling in the Time of Revolution." *Information Systems* 23, nos. 3–4: 127–155.

Nagl, M., and W. Marquardt. 2001. "Tool Integration via Cooperation Functionality." In *Third European Congress of Chemical Engineering (ECCE 3)*, paper 6-5. Available at ⟨http://www.dechema.de/veranstaltung/ecce/cd/toc.htm⟩.

Patzak, G. 1982. *Systemtechnik—Planung komplexer innovativer Systeme*. Berlin: Springer.

Perry, H., and C. H. Chilton. 1984. *Chemical Engineers Handbook*. New York: McGraw-Hill.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall.

Rumbaugh, J., I. Jacobson, and G. Booch. 1999. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley.

Technischen Güte und Lieferbedingungen (TGL). 1974. TGL 25000: Chemical Engineering Unit Operations—Classification. Departmental Standard of the German Democratic Republic, Berlin.

Uschold, M., and M. Gruninger. 1996. "Ontologies: Principles, Methods and Applications." *Knowledge Engineering Review* 11, no. 2: 93–136.

Contributors

Birgit Bayer
BASF AG
WLE/FB-A15
Postfach
67056 Ludwigshafen
Germany
⟨birgit-bayer@web.de⟩

Alex Borgida
Department of Computer Science
Hill Centre, Busch Campus
Rutgers University
New Brunswick, NJ 08903, USA
⟨borgida@aramis.rutgers.edu⟩

Mohamed Dahchour
Institut National des Postes et Télécommunications
Department of Informatics
Av. Allal Al Fassi
Rabat, Morocco
⟨dahchour@inpt.ac.ma⟩

Armin Eberlein
Department of Electrical and Computer Engineering
University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4
⟨eberlein@enel.ucalgary.ca⟩

Matthias Jarke
RWTH Aachen
Informatik V and Fraunhofer FIT
Ahornstr. 55
D-52056 Aachen
Germany
⟨jarke@cs.rwth-aachen.de⟩

Manfred A. Jeusfeld
Tilburg University
Department of Information Management
Warandelaan 2
Postbus 90153
5000 LE Tilburg
The Netherlands
⟨Manfred.Jeusfeld@uvt.nl⟩

Ralf Klamma
RWTH Aachen
Informatik V
Ahornstr. 55
D-52056 Aachen
Germany
⟨klamma@cs.rwth-aachen.de⟩

Kalle Lyytinen
Case Western Reserve University
10900 Euclid Avenue
Cleveland, OH 44106, USA
⟨Kalle.Lyytinen@case.edu⟩

Wolfgang Marquardt
RWTH Aachen
Lehrstuhl für Prozesstechnik
Templergraben 55
D-52056 Aachen
Germany
⟨marquardt@lpt.rwth-aachen.de⟩

John Mylopoulos
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 2E4
⟨jm@cs.toronto.edu⟩

Wolfgang Nejdl
Universität Hannover
Institut für Informationssysteme, Wissenbasierte Systeme
Appelstraße 4
D-30167 Hannover
Germany
⟨nejdl@kbs.uni-hannover.de⟩

Alain Pirotte
University of Louvain
IAG Management School (IAG)
Information Systems Unit (ISYS)
1, Place des Doyens
B-1348 Louvain-la-Neuve, Belgium
⟨pirotte@info.ucl.ac.be⟩

Christoph Quix
RWTH Aachen
Informatik V
Ahornstr. 55
D-52056 Aachen
Germany
⟨quix@cs.rwth-aachen.de⟩

William N. Robinson
Computer Information Systems Department
College of Business
Georgia State University
35 Broad Street, Suite 927
Atlanta, GA 30302-4015, USA
⟨wrobinson@gsu.edu⟩

Martin Wolpers
Fraunhofer FIT.ICON
Schloss Birlinghoven
53754 Sankt Augustin
Germany
⟨martin.wolpers@fit.fraunhofer.de⟩

Index

Note: The letter *t* following a page number denotes a table, the letter *f* denotes a figure, and the letter *n* denotes an endnote.