

Metamodeling for Method Engineering

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

The MIT Press
Cambridge, Massachusetts
London, England

© 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

1 A Sophisticate's Guide to Information Modeling

Alex Borgida and John Mylopoulos

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 model¹ 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-

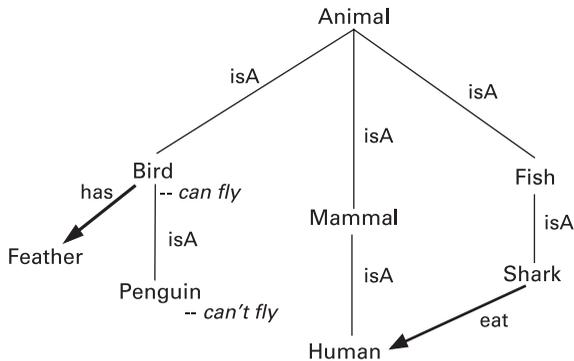


Figure 1.1
A simple semantic network

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

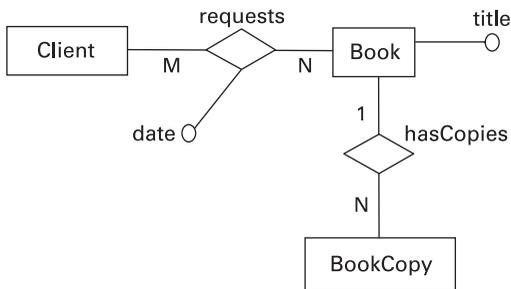


Figure 1.3
An entity-relationship diagram

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

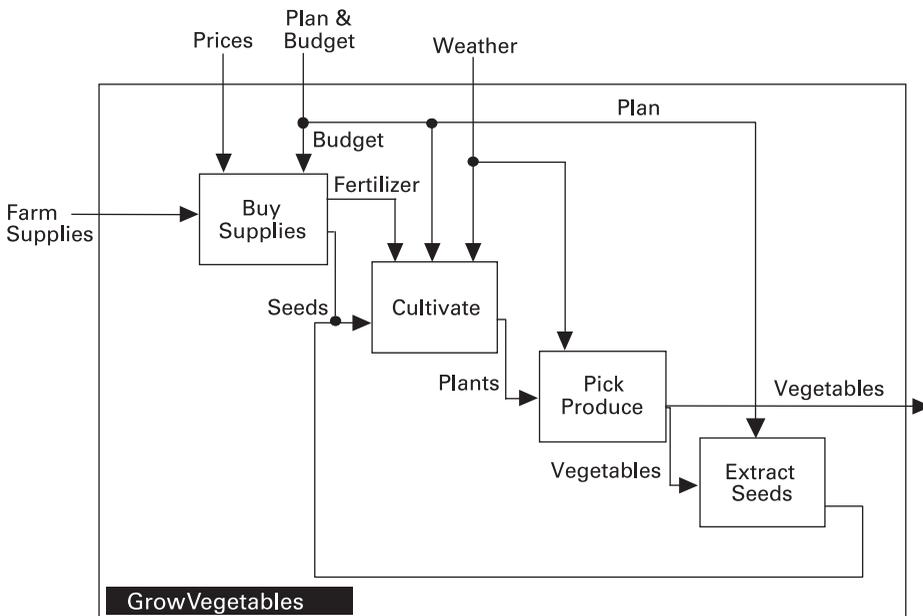


Figure 1.4
An SADT activity diagram

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 models⁴ 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. Tschritzis (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,

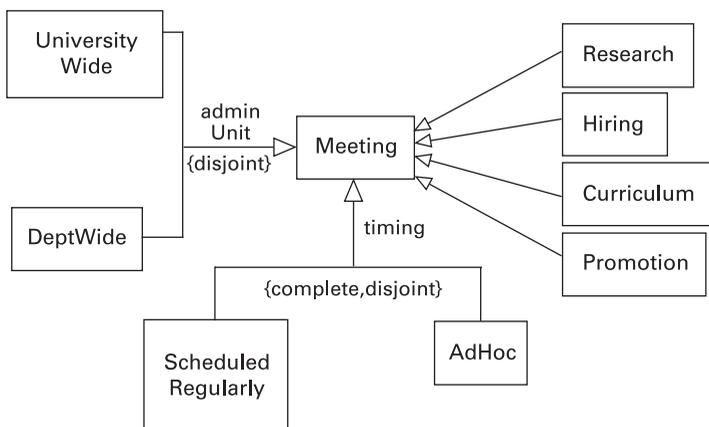


Figure 1.5
A subclass hierarchy for the class `Meeting`

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.

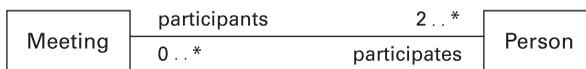


Figure 1.6

A semantic relationship between `Meeting` and `Person`

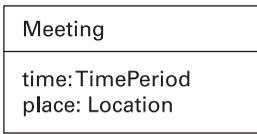


Figure 1.7
Attributes for Meeting

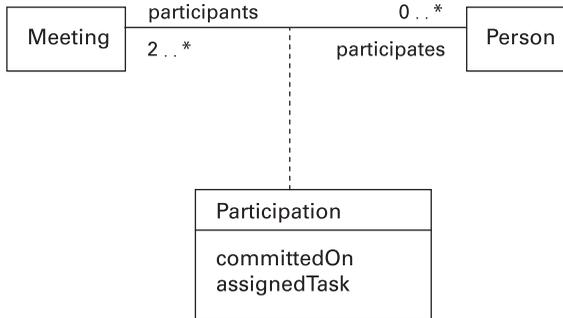


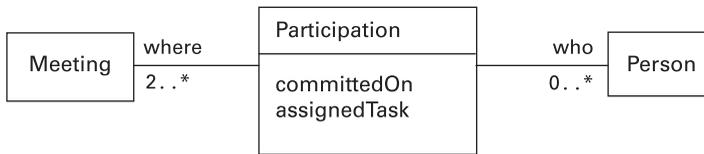
Figure 1.8
Participation as an association class

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


```
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.

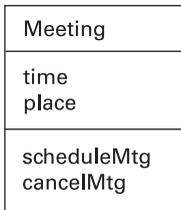


Figure 1.12
Some operations for `Meeting`

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 `ProvideConstr` 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.

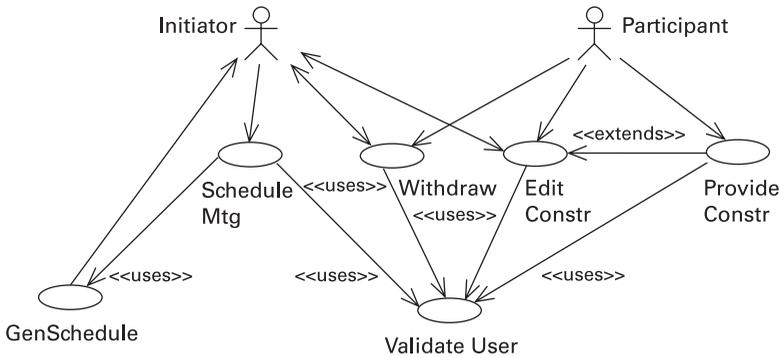


Figure 1.13
Use cases for a meeting-scheduling system

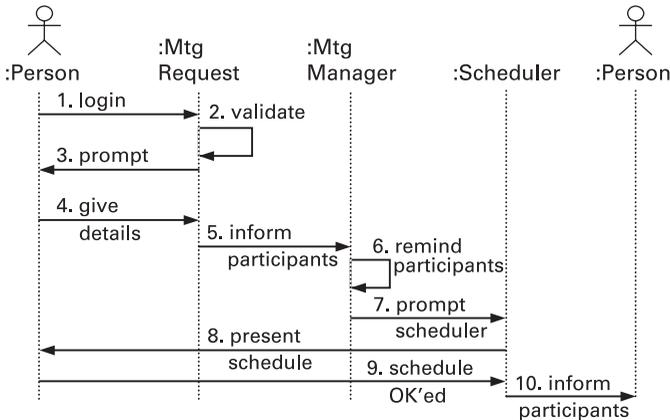


Figure 1.14
A sequence diagram for `ScheduleMtg`

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 scenarios, 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 \blacksquare `Scheduled(m)` means that `Scheduled(m)` held at all time points before now: $(\forall t:\text{Time}) \text{before}(t,\text{now}) \Rightarrow \text{Scheduled}(m,t)$. The expression $\blacksquare \leq 2\text{wk} \text{Scheduled}(m)$ 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

$$(\forall t:\text{Time}) \text{before}(t, \text{now}) \wedge \text{timeDist}(\text{now}, t) \leq 2\text{wk} \\ \Rightarrow \text{Scheduled}(m, t)$$

The notation \blacklozenge 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:

$$(\forall m:\text{Meeting})\text{Scheduled}(m) \wedge (\exists c:\text{"cancel"}) \text{Occur}(c) \\ \Rightarrow \text{OCancelled}(m)$$

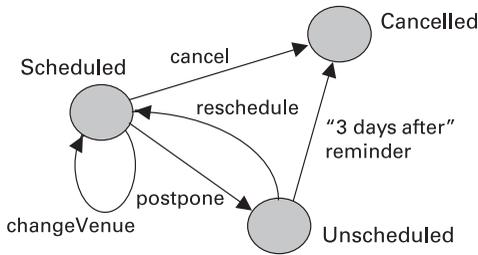


Figure 1.16
State transition diagram for meetings

The temporal operator \bigcirc 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-



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

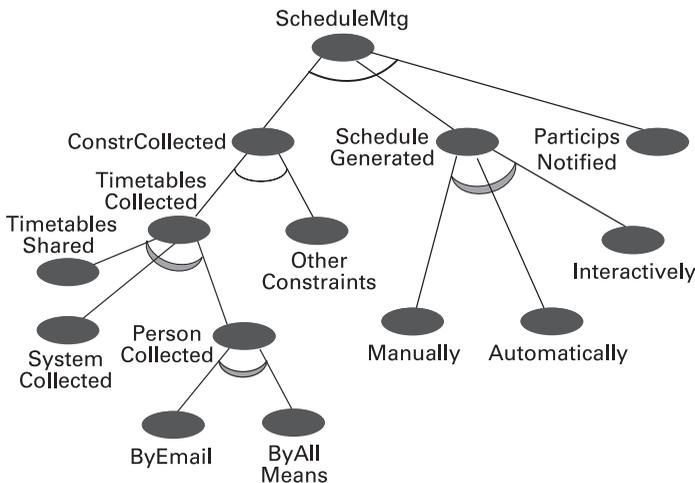


Figure 1.18
A (partial) space of alternatives for meeting scheduling

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 `ParticipisNotified`, 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, ParticipisNotified
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 *satisfied* 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 satisfy `UserFlexibility`; 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-

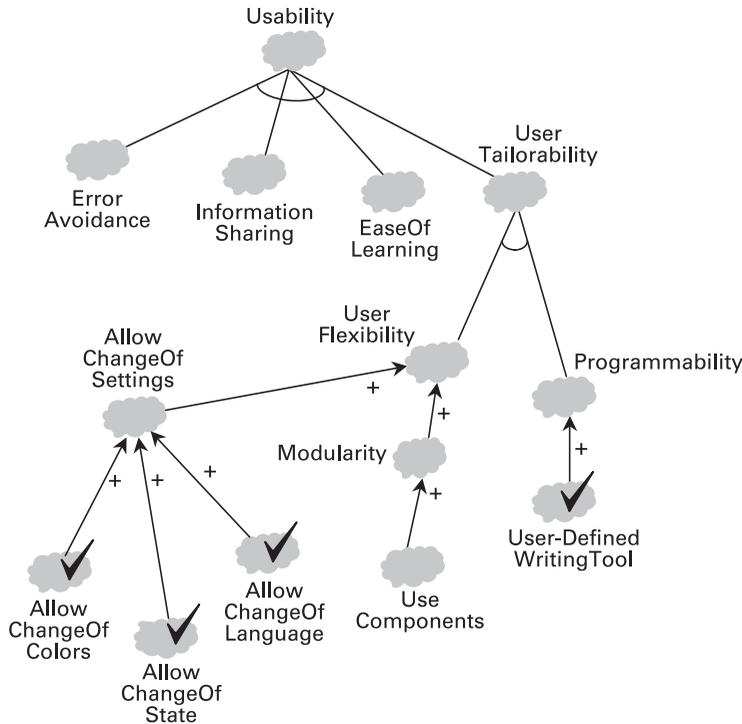


Figure 1.19
A (partial) softgoal hierarchy for Usability

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 satisfy 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 relationship⁸ between Performance

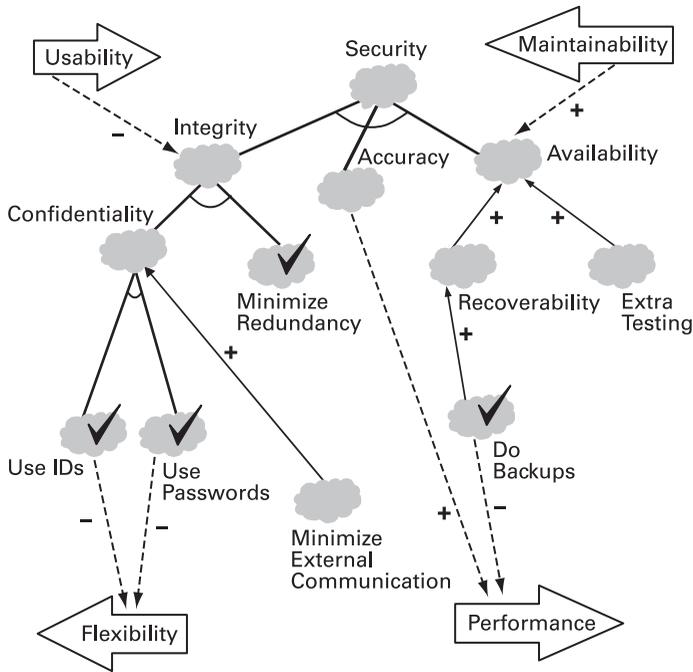


Figure 1.20

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

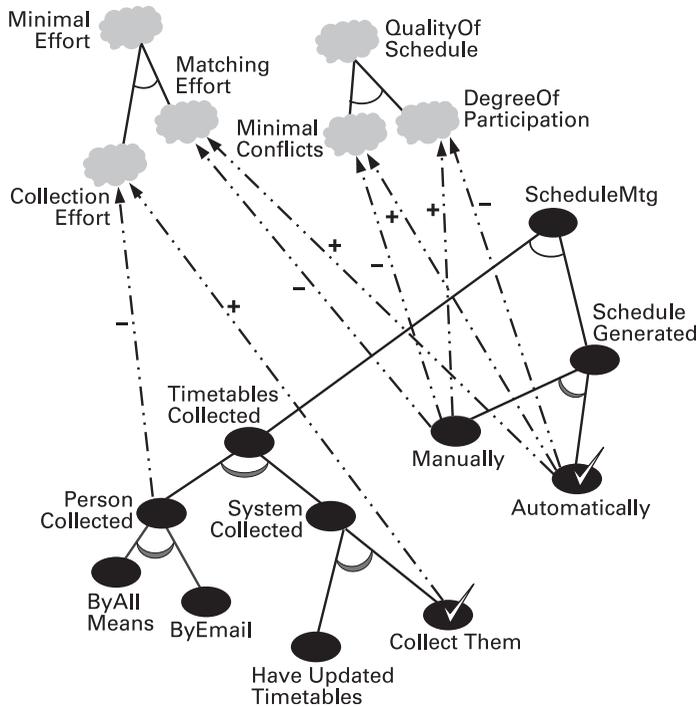


Figure 1.21
Evaluation of the goal `ScheduleMtg`

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 `QualityOfSchedule`. 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 *dependor* and the actor who is depended upon the *dependee*. For example, a professor (the dependor) 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 dependor. 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.

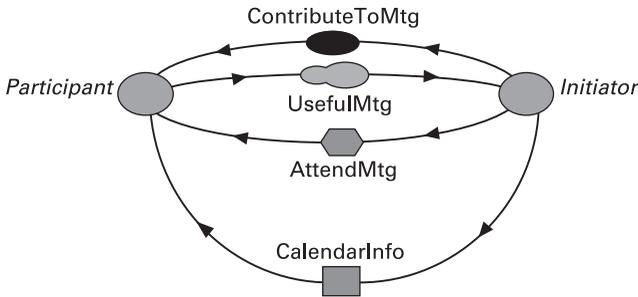


Figure 1.22
Initiator-participant dependencies for a meeting

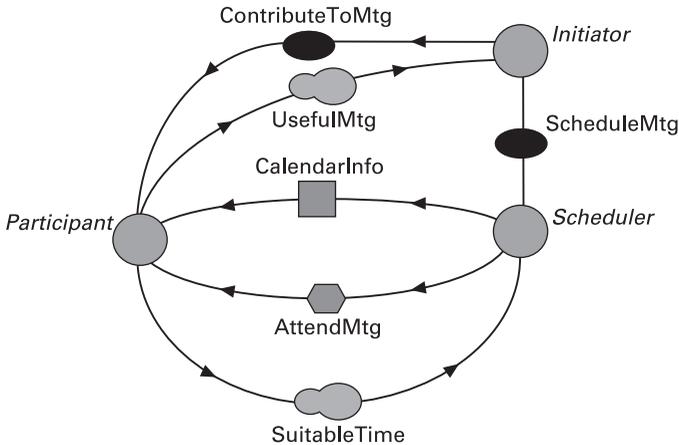


Figure 1.23
Initiator relies on scheduler to schedule meeting

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

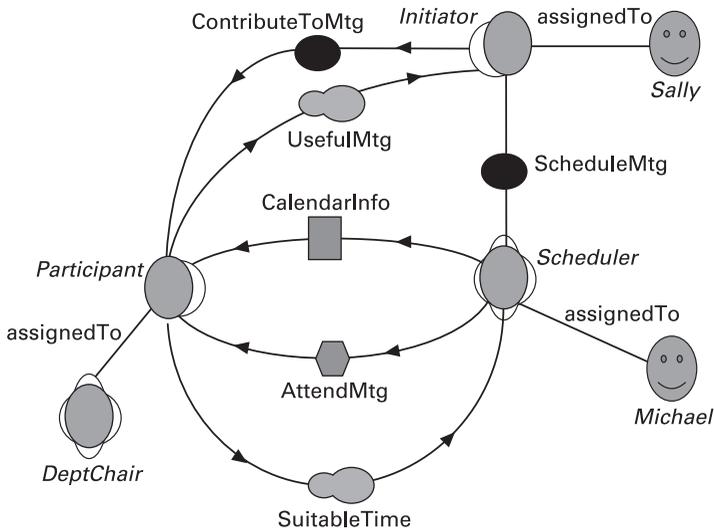


Figure 1.24
Agents, positions, and roles

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,

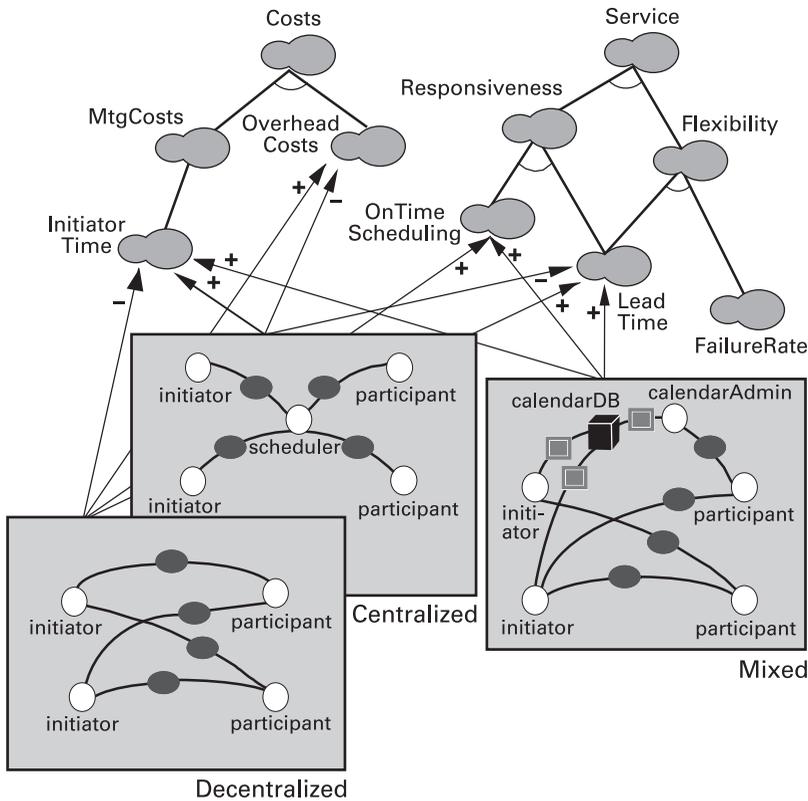


Figure 1.25
Evaluating alternative strategic dependency configurations

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. Kaggassala, 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. McGuinness, 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. Yorlmark, 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.