NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Towards Conceptually-Oriented Software Requirements Analysis and Design

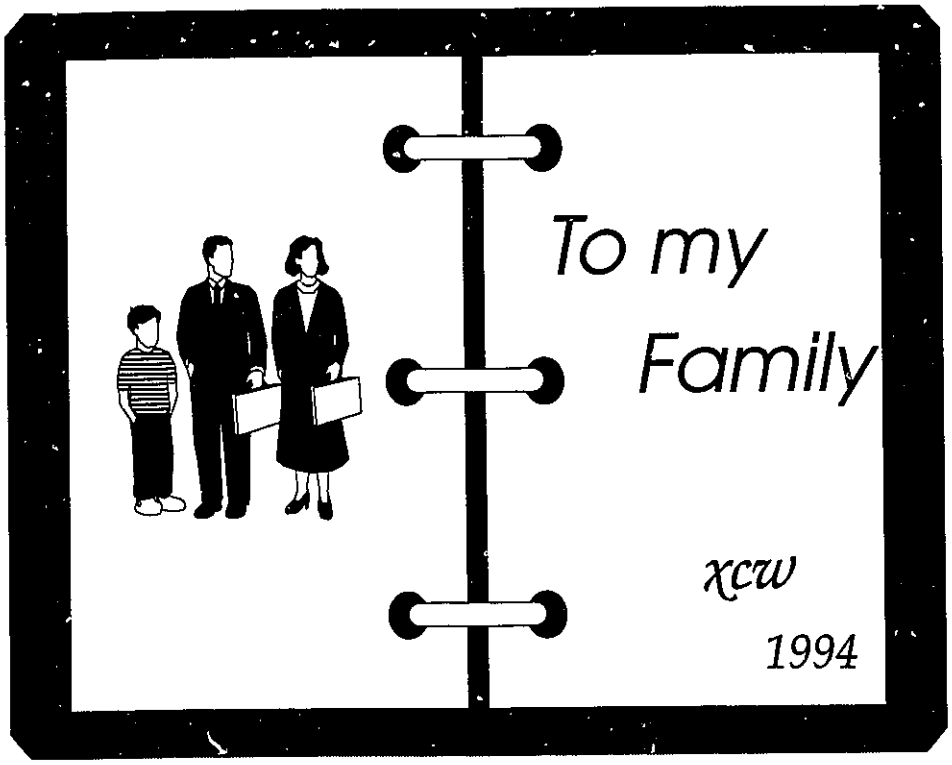
Xia-Cheng Wang

A Thesis
Submitted to the School of Graduate Studies and Research
in Partial Fulfilment of the Requirements
for the Master's Degree in Computer Science
Under the Auspices of
the Ottawa-Carleton Institute for Computer Science

Department of Computer Science
University of Ottawa
Ottawa, Ontario
Canada
May 1994

ISBN 0-315-96001-9

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA


To my
Family
xcw
1994

Abstract

Software developers currently do not have sufficient machine support for knowledge management. This thesis describes a partial solution to some of these knowledge management problems in software development, emphasizing conceptually-oriented requirements analysis and design, which we believe is fundamental to the software development process.

We propose an ontological framework which consists of a domain-independent ontology, a methodology ontology, and an application ontology. This framework permits multiple methodologies and applications to be integrated. The domain-independent ontology provides knowledge common to all domains, while the methodology ontology assists the designer to capture essential design knowledge following a specific methodology. The top level of the application ontology provides common structure over multiple applications. Lower levels of the application ontology correspond to each specific application, consistent with the domain-independent ontology and methodology ontology.

This ontological framework is supported by a generic knowledge management system known as CODE, which manages all knowledge in a knowledge base. A telecommunication example is developed to illustrate the approach. ObjecTime, an object-oriented commercial design CASE tool, is used to derive an executable design. Thus, we demonstrate that a knowledge management system (KMS) like CODE can be a powerful tool to provide intelligent machine support for conceptually-oriented analysis and design.

Acknowledgments

I am most grateful to Dr. Douglas Skuce for his never-ending encouragement, guidance and patience as my thesis supervisor. He has offered many constructive comments and suggestions at many critical moments during the course of this research.

I would like to thank Bran Selic of ObjecTime Limited for making the ObjecTime tool set available for this study. He has offered many suggestions and technical support about ObjecTime and acted as the telephony domain expert at some points during this study.

Thanks also goes to Dr. Hasan Ural for his guidance in the area of software testing where I obtained my knowledge of many software testing methodologies, and also for making the FrameMaker software available.

Many people in the AI laboratory also deserves a special thanks. To mention a few, Judy Kavanagh, Tim Lethbridge, Nagi Ghali are examples among the group.

I would also like to thank many of my friends for their encouragement, understanding, and friendship. I do not have to mention the names here. You know who you are.

At last, but not the least, I would like to thank my parents, my wife, and my son for their patience and support throughout my study. Without their understanding and encouragement, I do not think I could have finished this work.

Table of Contents

List of Figures

CHAPTER 1

Introduction

This chapter first provides a brief discussion of the relationship between software engineering and knowledge engineering. We focus on one of the major problems in the software development life cycle, which we believe is knowledge management. In particular, we are interested in what kinds of knowledge exist during the software development process, and how knowledge engineering techniques can assist software engineers to develop quality software more easily. The motivation and the objectives of the thesis is also outlined in this chapter, and an organizational sketch of the thesis is provided.

1.1 The Problem of Knowledge Management in Software Engineering

The software development life cycle, usually starts with the recognition, gathering, and understanding of the software requirements and proceeds with an attempt to specify, design and implement a system which satisfies the user's requirements ([TSEA92]). Any serious software development methodology, either structured or object-oriented, has to therefore deal with requirements analysis and specification. Typically the requirements analysis process requires the involvement of an application domain expert who is a specialist in the application domain but not necessarily a system expert, and one or more specification and design experts who are specialists in system design but not necessarily an application domain expert. The system specialists analyze the system requirements according to the user's requirements and information provided by the domain expert. Based upon this process, a requirements specification can be derived.

Most requirement specifications today are still written in natural language and vary in length from a few pages to five thousand ([DAV90]). However, the larger the natural language document, the more difficult to keep it consistent, to maintain it, and to understand it. One of the major problems with requirements in natural language is that it is often ill-defined, diffuse, inconsistent, and ambiguous. Nevertheless, software today is still almost totally defined by requirements and specification written solely in natural language.

Although there are some CASE tools now available to help software engineers carry out the designing task, there have been few, if any, tools or techniques that have been widely used in industry specifically to assist engineers in the requirements analysis phase of the software development process, the so called "upstream", as it represents the early stage of the software development. During most of the history of software engineering, this lack of upstream support was acceptable. The engineers simply juggled requirements as best they could, designed specifications that would meet the requirements as best they could, and turned the project over to those responsible for implementation. This type of software engineering, however, is facing great challenges by the demand for large and complex application systems in industry.

Usually these systems are distributed across networks, and are domain specific. Many different applications may have to be integrated to meet a variety of needs ([BELA91]). A good example of such systems are telecommunication systems. The development of such distributed real-time systems requires immense complexity both of the systems themselves and of the process of designing them. The traditional paper and pencil approaches are no longer able to manage all the associated knowledge during the process. Therefore *knowledge management*, which deals with acquiring, representing, storing, retrieving, debugging, validating, sharing and communicating of knowledge, can become a major bottleneck in software engineering.

[GHAL93] provides a discussion of the major knowledge-related problems during the software development life cycle. The first of the major problems software engineers face is often the complexity of the application domain. Typical applications are too large and complex for any single individual to understand [WIRF90], therefore, software engineers have to spend a great deal of time to gain an overall understanding of the problem domain. The second problem is the management of software knowledge, which ranges from knowledge of the development methodologies and application domains to the knowledge of existing software systems. As argued by [GHAL93], this problem is rooted in the lack of tools and techniques for properly storing, representing, sharing and communicating knowledge, i.e., knowledge management tools. Another problem is that knowledge gained at one phase of the development process may not be transmitted to the next phase properly and becomes lost, therefore requiring rediscovery of knowledge during the software process. It is also widely recognized that there is insufficient reuse, both reuse of the process and the code. The problem is rooted in the lack of knowledge of the existing processes and systems.

Another problem in the software development cycle is inadequate machine assistance for software developers. As indicated by [AMBR91], existing tools supply only a small amount of automated assistance and tool integration. As software maintenance counts for the largest portion of software development cost, perhaps the most severe problems are traceable to inadequate knowledge available to the maintainers, usually because of poor documentation. Software documents are

frequently incomplete, out of date, ambiguous, and sometimes contain conflicts. In most case, maintainers have to spend much time to read and try to understand software code and its logic provided by software designers and implementors, and this is a very frustrating and time consuming process.

To deal with problems mentioned above, various approaches have emerged. In the next section, we provide a brief discussion of how knowledge engineering techniques may help software developers to overcome knowledge-related obstacles encountered during the software development life cycle.

1.2 Knowledge Engineering vs. Software Engineering

As software engineering focuses on the transformation of a set of requirements into the code of a program or application that meets the requirements, knowledge engineering focuses on the transferring of knowledge in various forms into a knowledge base. Thus the knowledge base can play the role of code, except that instead of being executable, most knowledge bases are intended for human use. During the software development life cycle, many kinds of knowledge are involved at different development stages. In this thesis, we partition the knowledge involved in a typical software development cycle into three types: the domain-independent knowledge, the application knowledge, and development methodology knowledge. The following is an incomplete list of knowledge in these types:

Domain-independent knowledge: The most generic knowledge which does not belong to any specific domain.

Application knowledge:

- *Domain knowledge*: knowledge about the application domain itself, with no consideration of design or implementation decisions.
- *Design knowledge*: knowledge about the system design and its rationale, with no consideration of implementation.

- *Implementation knowledge*: knowledge about the implementation of the design.

Development methodology knowledge:

- *Requirements analysis methodology knowledge:* knowledge about the methodology used to carry out the requirements analysis.
- *Specification techniques knowledge*: knowledge about the techniques used to accomplish the requirements specification.
- *Design methodology knowledge*: knowledge about the methodology used to carry out the design task.
- *Programming language knowledge:* knowledge about the programming language employed to implement the design.
- *Testing methodology knowledge*: knowledge about the methodology employed to accomplish the testing task on the design and implementation.

In a typical software engineering life cycle such as the waterfall model, the requirements analysis phase requires a great deal of domain-independent and application knowledge. To accomplish this task, one needs also a great of deal knowledge of the analysis methodologies involved, for example, the Object Modeling Techniques (OMT) as described in [RUMB91). During the specification and design phases, a large volume of design knowledge is required, both in application domain and methodology domain. The coding or implementation phase and the following debugging and testing phases require implementation knowledge, not only the knowledge related to an implementation language, data structure, and algorithms, but also the associated knowledge of debugging and testing methodologies and techniques. The final phase is the maintenance, which can require domain knowledge, design knowledge, as well as the implementation knowledge.

In today's software industry, most software is large enough to require a development team to develop it. Therefore, communicating and sharing knowledge between team members becomes essential to accomplish the development task. Meanwhile, agreement between software developers on concepts and terminology of the system under development is also an important but often overlooked problem.

These elements are the points where software engineering and knowledge engineering intersect: the software engineer becomes an application system designer, who must understand the application domain in order to select the best software and hardware for a design's implementation. Perhaps most importantly, the engineer must be able to work well in teams with others, because a task of this complexity requires the expertise and cooperation of a wide array of specialists in the application domain, in real time performance, in reliability, and in many other areas.

For years computer science has developed an important technology known as *data management.* As *data* is only a basic element of *information* which consists of signals that can be input to some system and that affects its behavior, *knowledge* is information that can be possessed by a system capable of "knowing", i.e. one that has some intelligence. Unfortunately knowledge management is so far underdeveloped, therefore it is a major challenge in computer science. *Knowledge engineering*, a branch of artificial intelligence, focuses on all aspects of knowledge management problems. In software engineering, how to manage the various kinds of knowledge and what kind of assistance that knowledge engineering might provide to the process of software development then becomes an important issue. Generally speaking, knowledge engineering can provide software engineering a great deal of guidance and assistance in three aspects: assisting management of application knowledge, assisting management of methodology knowledge, and assisting derivation of better documentation for both the software under development and the development processes. We elaborate these three aspects in more detail in the following paragraphs.

Firstly, typical application domain knowledge is often buried in often poorly-written and ambiguous natural language descriptions or in some expert's head. How to acquire the essential domain knowledge to form the requirements and then map these requirements into a formal specification or design is a crucial task in software engineering. So far there are no CASE tools that can help software engineers to carry out such a duty. As one of the most important aspects of knowledge engineering is how to acquire and present knowledge, it is clear that knowledge engineering may assist software engineers to capture and formulate the domain knowledge more easily and efficiently. Meanwhile, a knowledge management system (KMS), which is

specially designed to assist one or more aspects of knowledge management, may provide knowledge engineers and/or software engineers with better tools to accomplish their knowledge-related tasks.

Secondly, today's software often requires sophisticated and complicated methodologies to develop, especially for large real-time and distributed systems such as telecommunication systems. As knowledge engineering may help to manage the application domain knowledge, it also can provide software engineering with better management of knowledge associated with the domain of software engineering methodologies, techniques, and tools used during the software engineering practice. This is a very important aspect, especially for complex computer-aided methodologies such as ROOM[1] and its supporting CASE tool ObjecTime ([SELI91a], [SELI92]). Meanwhile, to map the domain knowledge acquired during the requirements analysis into a design using CASE tools such as ObjecTime, it is essential to provide knowledge management support either in conventional natural-language-based documents or using a structured knowledge base supported by a KMS, as we will illustrate in this thesis.

Thirdly, as we indicated in the previous section, poor documentation is one of the key problems during software development and maintenance. Knowledge engineering can provide software engineering valuable assistance in the documentation of the system under development, and the documentation of the development process itself. When a KMS is used, the knowledge base developed can serve as the most active and accurate kind of documentation. This type of documentation, in contrast with the traditional natural-language-based documentation, is much easier to store, retrieve, search, and update. As a result of better documentation, both software component and development processes can be more easily reused.

In summary, knowledge engineering techniques supported by a well developed and user friendly KMS can provide software engineers powerful means to handle most

1. ROOM stands for Real-time Object Oriented Methodology

of their knowledge management needs during software development processes. They can also provide a common communication ground for a better understanding of the knowledge associated with the software engineering techniques and methodologies, as well as the domain, design and implementation knowledge of the system under development. Another very important benefit is that they can provide better and accurate documentation of the system under development. As a result, it encourages and facilitates software reuse, including the system components and also the development processes, since it is often the case that lack of software knowledge prevents software reuse.

CODE[1], developed in the Artificial Intelligence Laboratory at the University of Ottawa ([SKUC91]), is a KMS which can assist software engineering in all three above discussed aspects. In this thesis, we will illustrate how CODE can be used to assist software engineers to manage essential knowledge during the upstream processes of software development and then to help map the requirements into an ObjecTime design. The CODE knowledge base can then serve as an on-line document for the system under development including requirements, specifications, and the design.

1.3 Motivation and Objectives of the Thesis

This thesis is based upon the belief that knowledge engineering can offer software engineering much assistance in solving some of the knowledge-related problems encountered during software development processes, as we discussed in the previous sections. In this section, we discuss the main objectives of the thesis and its motivation. In brief, the main objectives of the thesis are:

1. to illustrate how a KMS may be used to carry out a conceptual analysis of software requirements
2. to illustrate how a KMS may be used to map the software requirements derived during analysis phase into an object-oriented design based on ObjecTime

1. CODE stands for Conceptually-Oriented Description/Design/Documentation Environment

3. to demonstrate how the knowledge base established during the requirements analysis and design may serve the purpose of software development documentation
4. to conduct a conceptual analysis of some most important behavior-related concepts and several extensions and variants of the finite state machine (FSM) formalism

We now elaborate and discuss each of the above objectives and their motivations in more detail.

It is a common practice in the software industry that a software designer starts a design from requirements written in natural language text, sometimes accompanied with a specification using some formal specification methods. In the latter case the formal specification may be derived directly from the natural language description of the requirements. As we indicated earlier, it is inevitable that the natural language description of the system requirements contains ambiguity and inconsistency. The gap between the conceptual analysis of the requirements in natural language and the formal specification or system design becomes one of the bottlenecks during software development. To address this issue, the primary focus of the thesis will be on how to use a KMS to assist software designer to acquire, formulate, and manage the software requirements, i.e. to conduct a conceptually-oriented requirements analysis *before* a formal requirement specification or system design.

In many cases, software designers face a common problem: how to efficiently derive a design based upon a given requirements specification. To attack this problem, we will also demonstrate how a KMS can be used as an efficient and active assistant to the designer to apply the application domain knowledge acquired during the requirement analysis phase to conduct a design using an object-oriented CASE tool. Although a specific CASE tool, ObjecTime, is used in this study, the approach outlined in this thesis applies to any design methodology and CASE tool.

It is commonly agreed that software documentation is one of the most important aspects of the software development life cycle. In many ways the most critical problem occurs in software maintenance, which includes fixing existing bugs, upgrading the system to add new features, or adapting the system for slightly different

purposes ([DEVA90]). Documentation plays a key role in this process, since it is impossible to do maintenance on large systems unless the maintainer understands the existing system; without proper documentation such a task is almost impossible to accomplish. [DEVA90] argues that "A developer must spend a great deal of time discovering features of an existing system, ranging from the overall software organization and the conceptual framework that drove that organization to the location and details of specific functions and data structure. All of this is prerequisite to implementing the actual modification for which the developer is responsible". In this thesis, we will illustrate how a knowledge base derived during requirement analysis and design can function as a structured and active on-line documentation of the system under design.

Another problem in software development is the use of a formalism without common agreement on basic concepts or terminology. For example, the common Finite State Machine formalism (FSM) is one of the most widely used formalisms in the development of communication software to specify the dynamic behavior of these system under development. However, there is no common agreement about the behavior-related concepts in the literature. For example, the concept of "state" has many incompatible interpretations. In this thesis, we will illustrate a conceptually-oriented analysis of the important behavior-related concepts and several extensions and variants of FSM formalism.

In summary, the main objective of the thesis is to illustrate how a knowledge management environment, which consists of a KMS, an ontological framework, and conceptually-oriented analysis techniques, can manage the various knowledge types involved in a typical software development process and provide intelligent machine support to software developers by interacting with a software development environment (Figure 1.1).

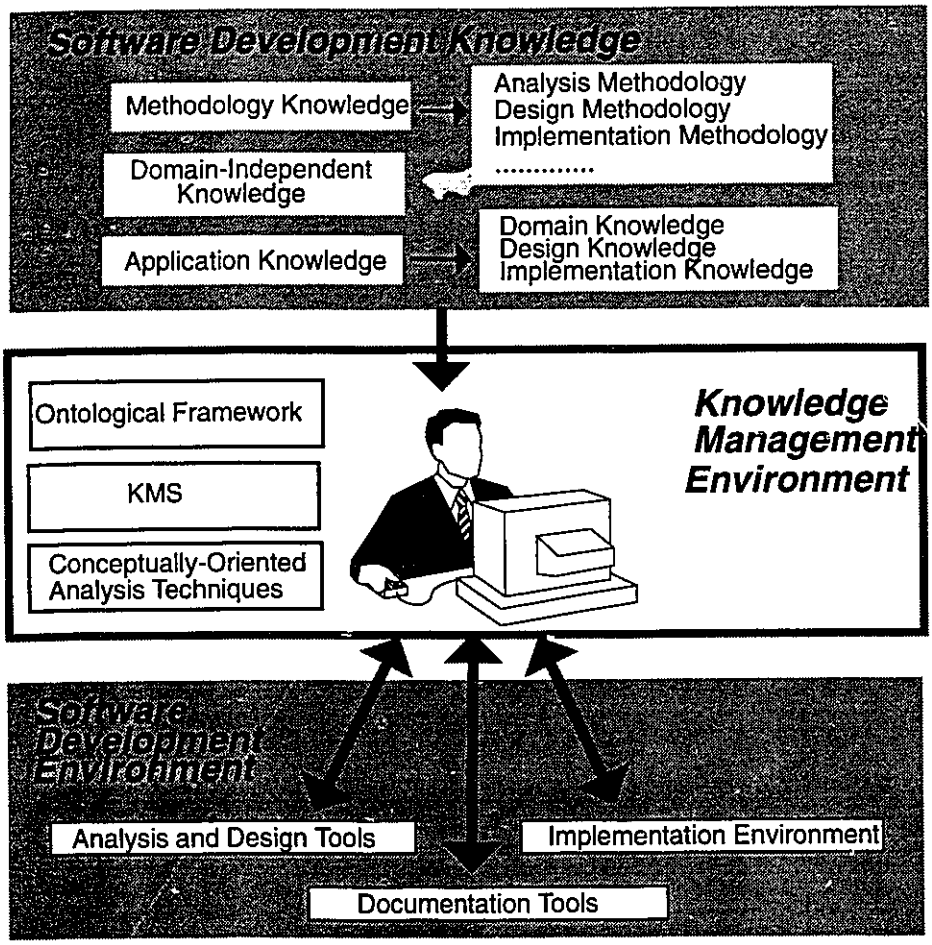


FIGURE 1.1 *Relationship between a knowledge management environment and a software development environment*

1.4 Organization of the Thesis

The following is a brief description of the remaining chapters of the thesis:

Chapter 2 provides a brief discussion of related work and examines some of the best current software development systems and techniques which provide knowledge-based assistance to the software engineer during the software development

process. We will provide a discussion of the principal approaches in these systems and techniques, as well as their advantages and shortcomings.

Chapter 3 provides a detailed description and discussion of the CODE and ObjecTime systems, upon which this thesis is based. The former is an object-oriented KMS, while the latter is an object-oriented CASE tool specifically developed for the design of object-oriented real-time systems such as telecommunication systems. We will investigate the basic principles and the major features of these two systems.

Chapter 4 will illustrate the use of CODE as an assistant to conduct a conceptually-oriented analysis of some behavior-related concepts and their ontology[1], which is used in ObjecTime to model the dynamic behavior of a system. Since "state" is the key concept in the FSM formalism, we will analyze this concept according to its basic characteristics and various usages. We will also investigate how other concepts in this formalism are used in various applications. Different representation mechanisms of the formalism will also be discussed.

Chapter 5 provides a detailed example of how a KMS such as CODE may be used to help the software designer to acquire application knowledge as well as methodology knowledge, how to conduct an ObjecTime design using the knowledge acquired during the requirements analysis, and how to use the knowledge base derived during the analysis and design process as an on-line active documentation. An example, requirements analysis and design of a simple telephone switching system, will be used in this chapter to illustrate how a conceptually-oriented approach and knowledge management mechanism in CODE can help the software designer to build quality software more easily and efficiently.

Chapter 6 provides a summary and conclusion of the thesis. A brief discussion of major contributions of the thesis will be provided. A short list of possible future work will also be presented.

1. Ontology troditionally means " a description of everything that exists", or, "a theory of existence". In this thesis we follow the definition of [SKUC90] as "the very general, usually tacit, basic semantic categories (e.g. entities, properties, situations) and their properties that people share".

CHAPTER 2

Knowledge-based Software Development Systems

This chapter provides a brief discussion of related work and examines some software development systems and techniques which provide knowledge-based assistance to the software engineer during the software development process. We will provide a discussion of the principal approaches in these systems and techniques, as well as their advantages and shortcomings.

2.1 Introduction

Considerable research is being done in the area of software engineering to simplify the development of software systems by providing software engineers powerful CASE tools and sophisticated design and programming environments. However, as indicated in chapter 1, software developers still require more assistance and guidance from more intelligent systems, and the key problem is the lack of knowledge management systems.

To meet the requirements of sophisticated software development, much research is being done in the area of developing intelligent knowledge management techniques and systems to provide software engineers more adequate support for software requirements analysis, specification, design, and documentation. Current research in this area may be characterized into two groups: 1) research in developing generic knowledge management systems, and 2) research in developing knowledge management techniques and tools specifically for software engineering. In this chapter, we focus only on the current research in the second group, i.e. our goal is to provide a brief overview of some knowledge-based software development systems.

2.2 Knowledge-Based Software Development Systems

As indicated in [DEBE91], the 1983 Knowledge-Based Software Assistant (KBSA) report ([GREE83]) described a new paradigm for software development, in which knowledge-based system assists a developer in all phases of the life cycle, from the initial capture of requirements to the generation of efficient code and testing the product. Since then a series of research projects have been conducted towards developing such systems. In this section, our discussion is mainly focused on systems specifically developed for managing software development knowledge in a knowledge base.

[SCHO88] proposed a knowledge-intensive development environment, which is based upon two key technologies: an object-oriented programming language known as Strobe, and a user interface framework known as Impulse-86. This knowledge-based assistant provides intelligent assistance for building knowledge-based systems. Ideally, the assistant should provide intelligent assistance in acquisition of an initial model of a problem domain, acquisition of control and task-specific inference knowledge, testing and validation, and long-term maintenance of encoded knowledge. The assistant requires five categories of knowledge in order to provide meaningful advice ([SCHO88]):

- knowledge of broad categories of application areas
- knowledge of knowledge base design
- knowledge of the semantics of the language used to encode the domain model
- knowledge of the editing tools available to the designer
- knowledge of how to infer the goals, plans, and intentions of the users

The first key technology in this approach, the Strobe language, is useful as the link that joins distinct software subsystems in a unified manner, and as a simple representation language kernel which supports the construction of computational models mirroring the organization of the physical world in which the software systems are to operate. The second key technology in this approach, Impulse-86, is a Strobe-based tool which supports user interface design. It implements the low-level mechanics of user-interfaces: support for windows, menus, providing editors and browsers for domain knowledge bases. It further serves as an extensible kernel from which application specific user interfaces can be constructed. However, this approach has several shortcomings. It does not help sufficiently in a team environment. It also can not generate complete documentation for the developed system. Also from the available material that it is not clear if the system can provide import/export assistance to transfer the knowledge captured during the design process from/to other systems.

More recently, [DEVA90] and [DEVA91] discussed the *complexity* and *invisibility* problems that is inheritent in the task of developing large software systems and described a knowledge-based system for software development, known as LaSSIE. The term *complexity* refers to the complexity of the development process, the application domain, the internals of the system etc. *Invisibility* refers to the fact that the structure of software is hidden, and the only external evidence we have of software is its behavior when executing. In many cases, there is no clear presentation of the architecture for programmers to examine as they extend the system. These problems complicate the task for acquiring the knowledge that a programmer needs to maintain a system, especially a large one.

To attack these problems, LaSSIE was developed to integrate architectural, conceptual and code views of a large software system into a knowledge base for use by developers, i.e. it can be used to formally encode software knowledge into a knowledge base so that it can be easily understood and retrieved when necessary. The LaSSIE knowledge base focuses on storing knowledge about a large real-time software system, and uses this knowledge as an index into a library of reusable components. It describes the functionality of the software from a conceptual viewpoint, with some information about its architectural aspects. It is built using a classification-based knowledge representation language, KANDOR, to provide classification and retrieval. The knowledge base serves as a repository of information about the system under development, as well as an intelligent index for re-usable components. To enhance its use for software engineering, it provides a user interface with a graphical browser and a natural language query processing system.

LaSSIE is based on a formal language for knowledge representation that features inheritance and classification, rather than a KMS. User interface and other modes of usage, which are essential for a KMS, are not its main concern. Its power is restricted by its high formality, which limits the expressiveness.

While LaSSIE emphasized the high-level "action and object" domain model of a telecommunication switch, the CODE-BASE ([SELF90]) project emphasized the need for comprehensive links of the domain knowledge to the code itself. CODE-BASE is a software information system that uses frame-based knowledge

representation technology to represent a wide spectrum of knowledge about telecommunication software. It is built on top of Classic knowledge representation system ([BORG89]) which provides inheritance, classification, contradiction detection and simple forward chaining.

CODE-BASE, as described in [SELF90], captures code knowledge about the Call Processing subsystem for a mid-sized AT&T private branching exchange (PBX) switch. Three categories of code knowledge are represented in CODE-BASE: the file and directory structure of the software base for the Call Processing subsystem, the definition and use of code objects, and the set of processes that make up the subsystem and the set of messages between these processes. The acquisition of this knowledge is automated through systems that extract, from C source files, the code objects, their relations with other objects and the places where they are used. This knowledge is then represented in a *is-a* hierarchy with inheritance. A graphical user interface is provided for knowledge access or retrieval with a specific query syntax.

CODE-BASE approach attempts to solve the problem of invisibility by linking the domain knowledge to the code itself. However, such a linkage is static rather than dynamic. Furthermore, the automatic classification limits the expressiveness in the knowledge acquisition process. Similar to LaSSIE, the formal language for knowledge retrieval also imposes inflexibility.

To address the issue of smoothing the transition between user needs, analysis, and design during the software development life cycle, [PUNC88] reports a knowledge-based software development environment called ASPIS, which includes a set of tools that specifically address tasks in the early phases of the software life cycle: requirements analysis and design. ASPIS consists of four assistants: an Analysis Assistant, a Design Assistant, a Prototype Assistant, and a Reuse Assistant. The first two are knowledge-based assistants which are used directly by the developers of a particular application adhering to a particular methodology, and the last two are support assistants. Once the knowledge about both the method and the application domain is defined, the specifications can be executed by the Prototype Assistant, which verifies the system's properties. The Reuse Assistant is used to help developers to reuse specifications and designs.

ASPIS exploits artificial intelligence techniques in a software development environment. It tackles issues of defining a formal specifications language, rapid prototyping, software reuse, and developing knowledge-based tools that embody knowledge both about the methodology and the application domain.

In general, ASPIS is more a set of CASE tool than a KMS; it does not support most knowledge operations, such as knowledge acquisition, debugging, and inferencing. Meanwhile, since the analysis method supported by the system is the structured analysis method, it does not support the object-oriented approach, which limits its application in the OO world. Furthermore, the system does not provide a graphical user interface.

Another knowledge-based software assistant is the Lockheed Sanders' Knowledge-Based Requirements Assistant ([HARR88]), which provides facilities for acquisition of informal requirements, entered as structured text and diagrams. The Requirements Assistant allows users to describe systems from different points of view such as informal text, data flow, state transition, and functional decomposition. It maintains an internal representation for the system being built which integrates these different views. By recognizing words in a lexicon of domain concepts and by providing hypertext-like support, it assists in the formalization of informal text.

A similar system, known as Knowledge-Based Specification Assistant ([JOHN88]), provides an environment for development of evolution transformations for specification modification. It also provides tools to evaluate the specification. One of these tools is a paraphraser which translates specifications into English. Another tool is a symbolic evaluator for simulating the specification and proving theorems about it. Some static analysis tools are also provided to automatically maintain and update analysis information as the specification is transformed.

Based upon earlier efforts of Requirements Assistant and Specification Assistant, [JOHN90], [JOHN91] and [JOHN92] report a requirements / specification environment called ARIES (Acquisition of Requirements and Incremental Evolution of Specification). The system is intended to be used by requirements analysts for evaluating system requirements and codifying them in formal specifications. As

formal specification languages are difficult to use in requirements acquisition, particularly by people who are not experts in logic, ARIES provides tools for the gradual evolution of acquired requirements, expressed in hypertext and graphical diagrams, into formal specifications. As an intensively knowledge-based system, ARIES incorporates knowledge about application domains, system components, and design processes, and supports analysts in applying this knowledge to the requirements analysis process. The complexity management of the Requirement Assistant and the formal specification constructs of the Specification Assistant are integrated into a simple wide-spectrum representation in ARIES, and the key capabilities of the Requirements Assistant and Specification Assistant are combined.

ARIES allows for multiple versions of the same concept to coexist, while capturing their similarities. It also allows individual projects to choose which version of a shared concept will be employed in their project.

A set of tools are provided in ARIES to provide assistance in the requirements analysis and specification processes. The acquisition tools are provided to capture initial statements of requirements as simply and directly as possible. The analysis tools, including a constraint propagation engine and an incremental static analyzer, are provided to check for completeness and consistency. The simulation tools are provided to determine appropriate parameters for requirements or to discover unexpected or erroneous behavior, i.e. to observe the behavior of a proposed system or its environment.

Similar to the approach of ARIES, the Programmer's Apprentice (PA) project ([RICH89], [RICH92], [REUB89], [REUB91]) aimed to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify, and document programs, and to build a tool which will act as an intelligent assistant to a skilled programmer. Two intermingled lines of activities were carried out during the project. One is to develop knowledge representation and reasoning techniques, and another is to build prototype systems to demonstrate the feasibility of various kinds of programming automation. The project consists of three phases, yielded with three prototyping systems: the Implementation Apprentice, the Design Apprentice, and the Requirements Apprentice.

The cornerstone of the Programmer's Apprentice is a formal representation for programs and programming knowledge, called the Plan Calculus, which makes data flow, control flow, and the design of a program more explicit, as compared to program text or parse trees. The Requirements Apprentice uses standard artificial intelligence knowledge representation and reasoning techniques, i.e. frames and constraints, to detect inconsistency and incompleteness in informal requirements descriptions. A central aspect of this project is the codification of *clichés*, i.e. commonly occurring structures (combinations of the primitives) expressed as a set of roles and constraints. The roles of a cliché are parts that vary from one occurrence of the cliché to the next, while the constraints specify fixed elements of structure, i.e. the parts that exist in every occurrence, and are used to check that the parts that fill the roles in a partially specified occurrence of a cliché are consistent. clichés are organized into libraries by means of taxonomic hierarchies and overlays, and are used to support program synthesis and recognition of clichés in existing programs.

The Implementation Apprentice, also known as KBEmacs (Knowledge-Based Emacs), is a prototype developed to demonstrate the usefulness of the Programmer's Apprentice approach in the implementation phase of a software development process. It employs an user-extendable library of implementation clichés to support automatic implementation of a program once a software engineer has selected the appropriate algorithmic fragments to use. It allows the programmer to operate directly on the algorithmic structure of a program. The system also can create a comment explaining the implementation structure of a program.

The Design Apprentice uses design clichés, which embody knowledge about typical specifications, typical designs, and typical hardware, to assist a programmer in the detailed design of programs. Each of these clichés is annotated with information about what roles and constraints are mandatory, likely, or possible. The library of design clichés enables the programmer to describe the design of a program concisely. The Design Apprentice can detect and roughly explain errors made by the programmers, based upon a classification of possible incompleteness and incosistencies. It automatically select reasonable implementation choices, and

automates detailed design by automatically selecting appropriate algorithms and data structures. Programmers can use the Design Apprentice to change design decisions, introduce and remove instrumentation, add error checking, and make low-level additions to the code.

The Requirements Apprentice uses a library of requirements clichés to assists an analyst in the creation and modification of software requirements. The focus of the Requirements Apprentice is on the formalization phase that bridges the gap between an informal and formal specification. The apprentice attempts to overcome the problems of human communication, such as abbreviations, ambiguity, poor ordering, incompleteness, contradiction, and inaccuracy. It accepts a restrictive natural language input, and produce an interactive output to notify the analyst of conclusions drawn and inconsistencies detected while requirements information is being entered. A machine manipulable Requirement Knowledge-Base (RKB) is maintained to represent everything the Requirements Apprentice knows about the evolving requirements. A requirements document can be created by summarizing the RKB.

In summary, the Programmer's Apprentice captures three important types of software development knowledge in the form of clichés, i.e. the requirement knowledge, design knowledge, and implementation knowledge, and use these knowledge to provide knowledge-based assistance to the software developers. The clichés library promotes software reuse. However, it did not provide a good user-friendly interface in knowledge acquisition and retrieval, and the users are forced to use exactly the same names and terms known to the system in the process of knowledge retrieval during these three phases of development.

2.3 Summary

In this chapter we surveyed a number of knowledge-based software development tools and environments, which have features to our research. In summary, all these systems were developed particularly to provide knowledge-based assistance for software developers, and encode software knowledge acquired during the development process into a structured knowledge base. This knowledge then can be retrieved and manipulated in order to provide machine support for software development tasks.

There are certain common weaknesses in these systems. All these system are strong in some aspects of knowledge management capabilities, but weak in others. They also need enhancement in dealing with natural language related problems. Many of these systems have no direct link between the knowledge base and the programming environment, and the user expressiveness is limited due to the knowledge representation systems designed to support automatic inferencing. Poor user interface is a common problem in many of these systems. Some of these systems, such as LaSSIE, do not provide a wide spectrum of formality. Furthermore, the current generation of the knowledge-based software development systems supports only single-user knowledge bases. None of the systems which we reviewed in this chapter can provide support for the basic knowledge base sharing and locking mechanisms that allow multiple developers to work simultaneously in a team environment.

CHAPTER 3

The CODE and ObjecTime Systems

This chapter provides a detailed description of the CODE and ObjecTime systems, upon which this thesis is based. The former is an object-oriented KMS, while the latter is an object-oriented CASE tool for designing object-oriented real-time systems such as telecommunication systems. We will investigate the basic principles and the major features of these two systems.

3.1 Introduction

In chapter 2 we gave a brief overview of several well-known software development systems and techniques which provide the software designer with knowledge-based assistance. In this chapter, we first provide a detailed discussion of the principle knowledge representation mechanism in a generic knowledge management system, CODE, then we provide a detailed discussion of an object-oriented design methodology, ROOM, and its supporting CASE tool, ObjecTime. These two systems are used in this thesis to investigate knowledge management techniques in software development.

3.2 The Conceptually-Oriented Development Approach (CODA) and CODE System

This section describes the basic concepts in CODA and the associated CODE system. The former is an analytical approach, and the latter is a tool which provides intelligent machine support to one or more analysts, or persons seeking knowledge.

3.2.1 CODA and Conceptual Analysis

The object-oriented development approach stresses the recognition of objects and their responsibilities, while CODA ([SKUC93b]) stresses the need for identifying concepts and conceptual / linguistic analysis. CODA differs from the object-oriented paradigm in that concepts are more generic than objects. For example, On an object-oriented approach, we would not consider an event as an object, but it certainly is an important concept and may need much analysis. Therefore, CODA is more generic than the traditional object-oriented approach.

The term *conceptual analysis* means to clarify which concepts and their relationships are needed to describe something ([SKUC93c]). The target of the analysis process is to derive a clear understanding of some subject, and describe it in a consistent and unambiguous manner. The vehicle used in this process could range

from very informal methods such as natural language to very formal approaches such as mathematical equations or logic. In CODE, a concept may be described by structured and semiformal property hierarchies which may denote either attributes or relationships to other concepts. We will describe this idea more detail in next section.

CODA is a collection of principles and techniques, supported by the CODE system, that are intended to assist in conceptual analysis and knowledge manipulation over a wide spectrum of applications. It integrates ideas from knowledge engineering, object-oriented programming, object-oriented analysis and design, natural language, and logic ([SKUC93c]).

CODA is based upon a top level ontology. The term *ontology* means a description of everything that exists, or a theory of existence. It is sometimes also used to mean all very general ideas, concepts, principles, etc. used in structuring knowledge. It is also often used in knowledge engineering to mean some particular knowledge base constructed following some general ontological guidelines. In CODA, it means the most general things, concepts, properties, principles, etc. needed to analyze some subject ([SKUC93a], [SKUM90], [SKUM91]).

3.2.2 CODE

The CODE system, a generic KMS, addresses general needs of conceptual and linguistic analysis and provides an intelligent machine support to a large user community such as knowledge engineers, software developers, linguists, etc. This section describes the main features and shortcomings of this system.

CODE has been evolved into four versions. The system described and used in this thesis is version 4, the latest. Instead of using term CODE4, we prefer to user term CODE as a general symbol to represent the system, since many features and key concepts are common in the different versions.

3.2.2.1 Basic Concepts and Terminology

CODE comes with an ontology of some fifty semantic categories somewhat analogous to the general classes in an object-oriented environment ([SKUC91]). As it is not desirable to introduce the complete CODE ontology and terminology here, we describe in this section some of the key concepts and terminologies we will use in this thesis. However the use of these is optional; the user may discard any s/he feels not needed, or may change them.

Thing and Concept

The top of CODE ontology is called '*thing*', i.e. anything we wish to discuss, describe, or define by stating properties of it. In other words, everything one might desire to discuss and hence represent is a thing. The representation of knowledge about a thing in CODE is a *concept* for that thing. In general, concepts correspond to anything we want to say something about, and are arranged in hierarchies in CODE. Concepts are denoted by terms or symbols which are either natural or formal language expression ([SKUC93c]). Concepts are either types or instances. Every concept corresponds to one thing, either an instance or a type. All concepts are represented by their properties. CODE organizes knowledge about each concept into *a conceptual descriptor*, which are analogous to frames in artificial intelligence or classes in object-oriented programming languages. Moreover, CODE supports property hierarchies, which is a feature differing from most existing knowledge management systems. We will provide examples of all these features later.

Properties

Each thing is characterized by its properties. In CODE, properties are divided into a fixed set of system properties and an open-ended set of user properties. The former controls basic operations like naming conceptual descriptors and inheriting properties. The later is provided by the user to define the thing under consideration. Every property for a given concept has a property name and a property value, which may refer to some other concepts. Properties usually are denoted by either verbs involving the thing as subject, modifiers like adjectives that describe it, or relationships to other things that often are referred to by nouns.

Statement

A statement in CODE has at least two essential parts: the subject which refers to a thing, and the predicate which refers to a property of that thing. In other words, every statement is composed of at least one main predicate and its subject. Thus, a concept for X is all the statements having X as subject, plus their component statements, etc. Of course a concept can be as small as a single statement.

Facet

Facets ([SKUL93]) are small statements representing incremental additions to a statement. There are two main types: facets corresponding to arguments of the statement, and facets for additional secondary information. The variety of facets depends upon the degree to which it is desired to approximate the meaning of a natural language sentence. The commonly used facets include value, modality, quantifier, choice, and comment. *Value* is the second argument in addition to the subject in a CODE predicate, and often the value may be optional. *Modality* is a fixed set of keywords conveying modal information, such as necessary, sufficient, typical, optional, impossible, etc. *Quantifier* is a fixed set of keywords conveying quantificational and set information, such as all, some, one, none, a set etc. *Choice* is a fixed set of keywords indicating nondeterministic choice such as one of, some of, all of etc. *Comment* is an unprocessed comment to any statement.

Metaconcept

Metaconcepts are associated properties that concepts frequently require that do not inherit since they are not logically properties of the instances but of the concept itself. For example, the creator of a concept only belongs to this concept, and should not be inherited to any of its subconcepts or instances. Therefore "creator" should be captured as a metaconcept property. For each user concept, there is a unique metaconcept which is an instance of the type metaconcept and created by the system itself.

3.2.2.2 Knowledge Representation and Main Features

[SKUL93] provides a list of features desirable in a good knowledge representation scheme. It must allow flexible and sufficient expressivity, but not be too complex. It should provide property inheritance, and support inferencing. As many problems in knowledge formulation and retrieval involve choosing or understanding imprecise or non-standardized terminology, the representation should permit the use of ambiguous terms, synonyms, concepts without names, and facilitate frequent renaming. Thus it is desirable to have support for some natural language processing. Meanwhile, the representation should not require the user to adhere to a strict syntax and semantics. The user should be able to express knowledge in any way that "feels comfortable" ranging from informal natural language phrases to rules in a formal language such as first order logic. Therefore, a wide formality spectrum is ideal.

Based upon the above considerations, the CODE designers provided a representation scheme in CODE which combines ideas borrowed from frame-based inheritance systems such as KEATS ([MOTE88], [MOTR91]) and CYC ([LANE90a], [LANE90b]), conceptual graphs ([SOWA84]), and term subsumption systems such as CLASSIC ([BRAS85], [BRAM91]), favoring expressiveness over the ability to perform complex automatic inferencing ([SKUL94]). It is easy to understand by non-AI specialists. It provides a wide range of formality such that users can use the system informally to clarify preliminary or developing ideas, or more formally to make precise definitions or to automate checking consistency of value restrictions.

As stated in [SKUC92], [SKUC93a], [SKUC93b], [SKUC93c], and [SKUL93], CODE is an object-oriented knowledge management system with an advanced and user friendly user interface. It is specifically designed to meet the needs of interactive knowledge management, and intended to facilitate most aspects of knowledge management, such as initial formulation and clarification, debugging, reaching consensus, retrieving, and disseminating. Clearly, these features are very desirable for our purpose.

Descriptions of the main features of CODE can be found in [SKUC91], [SKUC93c], [SKUL94]. In this section, we provide a list from a different perspective.

Knowledge Organization Features

- *Frame-based inference engine*: CODE combines some of the most useful features of frame-based inheritance systems and conceptual networks. It favors expressiveness over the ability to perform complex automatic inferencing.

- *Object orientation*: As an object-oriented system, CODE is programmed in Smalltalk, and runs on all major platforms. All properties of any given concept are fully expressed in the property hierarchy of this concept. CODE also supports multiple inheritance.

- *Concept hierarchy vs. property hierarchy:* CODE supports two main kinds of hierarchical structures: concept hierarchies and property hierarchies. The former models the is-a relationship among all participating concepts. This is-a hierarchy provides a mechanism for taxonomic structuring of knowledge and property inheritance. The property hierarchy, also known as the predicate hierarchy, provides a mechanism to categorize the properties a given concept has, i.e. all properties are arranged in a separate hierarchy in which the partial order is interpreted as "implies". For example, "wheel" is a subproperty of "parts", hence if the concept "car" has "wheel" as its property then "car" must have "parts" also.

User Interface Features

- *Control panel:* CODE provides a control panel to controls all top-level parameters, and also the default parameters for various views. It also serves as an interface for knowledge base operation, such as renaming, saving, loading into memory, removing from memory, or open various browsers for the knowledge base selected. Multiple knowledge bases may reside in the memory at once.

- *Outline browser*: Outline browser displays information in a textual format. Each concept or property is displayed as a line of text in the related pane (subwindow). It also provides hypertext-like features to allow the user to display only what is desired at the moment, or to rearrange knowledge.

- *Various graphical mode:* CODE provides highly developed graphical features. It can open one or more graphical views of any hierarchy, either a

concept hierarchy or property hierarchy. It can also show the non-hierarchical property relationships between the given concept and all other related concepts. Users may directly work on any of these graphs to perform adding, deleting, reparenting, moving, reshaping tasks. Any individual node may be selected for opening a property pane.

- *User composable browsers*: CODE provides a very flexible browser system which allows users to compose their own display windows, i.e. one can select a concept or a property of a concept (either in textual mode or graphic mode) and then open up another pane to display some other information. The user can open this pane as a separate window, driven by the current window, or s/he can attach the new pane to the existing window. If it is desirable, one may mix both graphical and textual panes into the same window.

- *Dynamic vs. static subwindow:* Sometimes it is desirable to have the subwindow dynamically driven by the master window, which means if the selection in the master window is changed, the contents of the subwindow should also be changed automatically. In other situations, it might be desirable to have a static display of some information for a given concept, which means when the selection in the master window is changed, the content of this static subwindow will remain unchanged.

Knowledge manipulation features

- *Database-like query operation:* CODE provides a database-like searching mechanism to answer various queries. It can search any text strings existing in the CODE knowledge base as either a concept or a property name. It also provides a masking mechanism to answer queries such as "show me all concepts which have property X" or "display all concepts which do not refer to Y".

- *Easy knowledge debugging:* CODE provides mechanisms for easy viewing of changes in property inheritance, and the system will warn of potential conflicts.

- *Property comparison matrices*: Property comparison matrices are a special feature designed to compare several concepts in terms of their properties. One may compare these concepts by selecting those properties of interest,

perhaps all. This feature makes it very easy to see how these concepts are similar or different.

- *Flexible formality*: In CODE, the degree of formality can be varied according to the user's preferences. A CODE knowledge base can capture knowledge ranging from extremely informal, e.g. unstructured natural language, to very formal, e.g. some version of logic. Along with an increase of formality, the system's ability to provide syntax and semantic checking can be increased.

- *Generality vs. speciality*: Although much attention has been given to the applications of the system in software engineering, the system is actually a general-purpose KMS. It can support knowledge management in any subject area. As examples, CODE has been used to build knowledge bases in various disciplines, ranging from computer sciences such as programming language or software engineering, linguistic analysis, to classification of rock types in geology.

- *Document processor:* With the assistance of a document processor, CODE permits users to process on-line documents line by line and directly transfer desired information into a coupled knowledge base. As stated in [SKUL94], the main purposes of this feature are to discipline the user's thinking so that each noun and verb is given attention, to permit verifying the knowledge base by inserting pointers from statements in the knowledge base to the sentences from which they were derived, and to eliminate the need to retype many phrases.

- *High inferencing capabilities*: CODE supports several inferencing mechanisms, which include inheritance, delegation, and an off-line full first order logic system (FOLDE). With these mechanisms, CODE can perform forward/backward chaining, contradiction and inconsistency checking, and semantic error detection.

- *Knowledge acquisition with ClearTalk*: As stated in [SKUC93c], ClearTalk is a simple, English-like language that allows the use of actual noun and verb phrases, including whole sentences, in CODE. It has a simple, unambiguous syntax, along with its unambiguous semantic interpretation, therefore, allows the system to provide considerable assistance in detecting semantic errors. It can be used simply as a restricted syntax, which allows

users to enter English-like statements and have the syntax and vocabulary checked so that they are constrained to use only very simple constructions.

3.2.2.3 Shortcomings

Beyond the developer's wish list, we present here some of the shortcomings from the user's viewpoint.

- *Lacks support for concurrent access of knowledge base:* Currently CODE does not support concurrent access of the same knowledge base. However, any serious application today, especially those in software engineering, can not be developed by a single person, and requires a strong teamwork environment. Therefore, lack of support for concurrency is one of the biggest shortcomings of the system.

- *Lacks rule-based inferencing*: Like most frame-based systems, CODE does not support rule-based inferencing, therefore, it is sometimes difficult to capture and check cause-effect relations and execution conditions. Although we may express such information through facets or metaconcepts, it is impossible to have semantic checking on these conditions.

- *Difficult to express execution sequence:* It is desirable sometimes to illustrate an execution sequences such as one statement becomes true only when the one before is true, especially in software requirements specification and design processes. As a user, the author of this thesis experienced difficulties to do so with CODE, although it is possible to express such relations informally by employing facets or metaconcepts features.

- *Weak information export to and import from other systems:* As CODE has been developed with a strong and flexible user interface, it does not provide a flexible import / export system so that knowledge captured with CODE may be easily exported to other systems, or so that necessary knowledge can be imported from other systems into a CODE knowledge base. For software engineering application in particular, it would be desirable if CODE can provide a easy linking mechanism so that it can be linked to a CASE tool such as ObjecTime (ObjecTime will be introduced in detail in

the next section). From the user's point of view, such a feature is very desirable, especially in an industrial environment.

- *Weak document creation system:* CODE provides only a simple output format so that users can export the knowledge base to a disk file and hence obtain a hard copy. However, this format is limited in flexibility and often is not what is desired. It would be desirable if it could transform part of or the whole knowledge base into a natural language document and provide the user with a flexible output format.A KIF-like[1] syntax and semantics, a notation and semantics for an extended version of first-order predicate calculus ([GENE92]), would be desirable for the publication and communication of knowledge ([GRUB93]).

- *Better syntactic and semantic checking:* Currently CODE provides syntactic and semantic checking in a limited degree. However, it is still heavily dependent on the user to check for possible conflicts or inconsistencies in a knowledge base. It would be desirable if more machine checking were available so that warnings can be provided when such a problem is possibly encountered.

3.3 The ROOM Methodology and ObjecTime

ROOM, a Real-Time Object Oriented Modeling methodology, is a new methodology for the development of real-time systems and is supported by a commercial CASE tool set, ObjecTime ([SELI94]). This methodology is founded on a set of principles which facilitates the rapid and efficient production of high-quality software ([SELI92]). In this section, we provide a high level overview of the methodology and its supporting tool set. First, we describe the conceptual framework of the methodology, then some basic concepts of domain modeling are discussed. A CODE knowledge base is built to clarify and classify these key concepts, and it is used to facilitate an ObjecTime design in a later chapter.

1. KIF stands for Knowledge Interchange Format

3.3.1 The Conceptual Framework of ROOM

The conceptual framework of ROOM is based upon two complementary modeling paradigms: the modeling dimension paradigm and the abstraction level paradigm. The former consists of three modeling dimensions: structure, behavior, and inheritance. The structure dimension captures the architecture of a system. It defines the components of a system and their communication and containment relationships. This dimension describes the static aspects of a system. The behavior dimension, in comparison with the structure dimension, captures the dynamic aspects of a system, i.e. the responses of each component of the system to any internal or external stimuli. Although many techniques may be used, the tool set employs a variation of the FSM formalism, known as ROOMCharts, to describe the behavior of a system. The inheritance dimension provides a basis for both abstraction and reuse. As ROOM organizes the complete system architecture in an inheritance hierarchy, it allows abstraction and reuse to occur at a much higher level than is possible with conventional programming languages.

The abstraction level paradigm also includes three parts: the system level, the concurrency level, and the detail level. This paradigm classifies the modeling concepts according to the scope which they encompass. The system level contains concepts such as layering which encompass the entire system, while the concurrency level deals with high-level abstractions, such as communicating FSM, which models the concurrency particular to the real-time domain. The detailed level contains concepts that are found in traditional non-concurrent programming languages, such as abstract data types and other implementation details.

3.3.2 Basic Concepts of Domain Modeling

The basic domain modeling concepts of ROOM can be organized according to the conceptual framework described in the previous section [SELI91b]. In this section, we provide a high level conceptual analysis of some basic ROOM modeling concepts using this framework, since we believe that these concepts are very important for any one who wishes to conduct an ObjecTime design. CODE was used to facilitate the conceptual analysis, i.e. a knowledge base about these basic concepts was built.

Actors

The most important concept at the concurrency level is actor. An *actor* models an active concurrent entity with well-defined responsibilities. Its implementation is hidden by an encapsulation shell. Figure 3.1 summarizes the basic properties for actor we defined in a CODE knowledge base. We summarize these as follows: Actors can be connected through special objects called *bindings,* which is an abstract communication path used to carry messages between actors, to form a larger system component. In other words, an actor can be structurally decomposed into a set of component actors with each of them performing a subset of the overall behavior of this actor. Such a decomposition can be applied to the component actors to an arbitrary degree. Therefore, an actor may have a set of component actors and a set of containing actors. Actors communicate with each other through a set of "openings" on its encapsulation shell called ports. The dynamic behavior of an actor is described by a FSM.

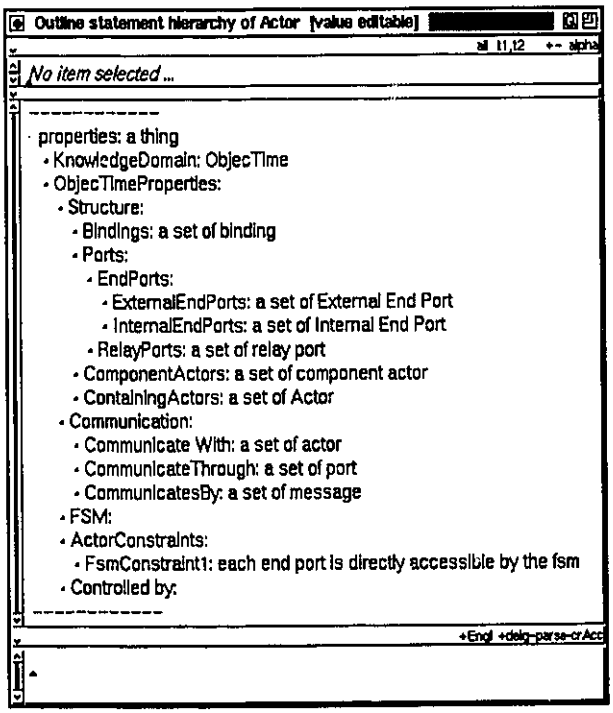


FIGURE 3.1 *Properties of Actor defined in a CODE knowledge base*

As illustrated Figure 3.2, the concept actor can be classified into a number of subconcepts according to their properties. Each of these subconcepts inherits all the properties from the concept actor. As CODE provides a "property comparison matrix" feature to retrieve those properties defined to the selected concepts and illustrates them in a matrix format, we will employ this feature to compare these subconcepts (Figure 3.3) to avoid a lengthy description.

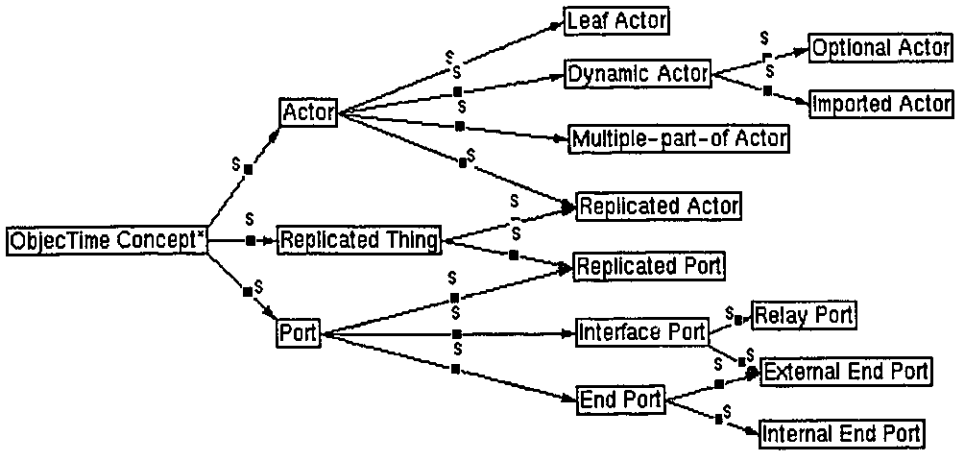


FIGURE 3.2 *Classification of concepts actors and port*

Leaf Actor: A leaf actor is an actor which has no component actors, i.e. it can not be decomposed further. Therefore, it has no bindings as part of its structure. It inherits all other properties from the generic concept of actor.

Replicated Actor: A replicated actor inherits all properties from the generic concept actor. It differs from other types of actor to facilitate designs in which multiple references to a common actor class are required repeatedly according to some regular pattern. It is replicated at design time.

Multiple-part-of Actor: The only difference that a multiple-part-of actor has from the generic concept of actor is that it must be contained in more than one actor, i.e. it is part of the decomposition structure of multiple actors simultaneously.

Matrix of statements of (Leaf Actor, Replicated Actor, Multiple-part-of Actor, Dynamic Actor)

	Leaf Actor	Replicated Actor	Multiple-part-of Actor	Dynamic Actor
ReplicatedAt	n/a	design time	n/a	n/a
MultipleReferenceTo	n/a	a common actor class	n/a	n/a
Bindings	na	a set of binding	a set of binding	a set of binding
ExternalEndPorts	a set of External End Port	a set of External End Port	a set of External End Port	a set of External End Port
InternalEndPorts	a set of Internal End Port	a set of Internal End Port	a set of Internal End Port	a set of Internal End Port
RelayPorts	a set of relay port	a set of relay port	a set of relay port	a set of relay port
ComponentActors	na	a set of component actor	a set of component actor	a set of actor
ContainingActors	a set of Actor	a set of Actor	more than one Actor	a set of Actor
Communicate With	a set of actor	a set of actor	a set of actor	a set of actor
CommunicateThrough	a set of port	a set of port	a set of port	a set of port
CommunicatesBy	a set of message	a set of message	a set of message	a set of message
FsmConstraint1	each end port is directly accessible by the fsm	each end port is directly accessible by the fsm	each end port is directly accessible by the fsm	each end port is directly accessible by the fsm
Controlled by	*	*	*	the containing actor
KnowledgeDomain	ObjecTime	ObjecTime	ObjecTime	ObjecTime
properties	a set of thing	a set of thing	a set of thing	a set of thing

FIGURE 3.3 *Property comparison of some subconcepts of actor*

Dynamic Actor: Dynamic actors are actors that are not automatically instantiated when their containing actor is created. They are only created / included or destroyed / excluded under the control of their containing actor at run time. There are two types of dynamic actors: optional actors and imported actors. An *optional actor* is an actor which is newly created by its containing actor at run time, and can be destroyed by its creator at run time too. An *imported actor* as its name indicates is an

existing actor which is included in its containing actor's decomposition at run time. Rather than destroy it, the containing actor can only exclude an imported actor from the containing actor's decomposition at run time.

Ports

Port is another important concept at the concurrency level. Each port on an actor represents a specialized interface of the actor. Figure 3.4 illustrates the properties of ports as we defined them in the knowledge base.

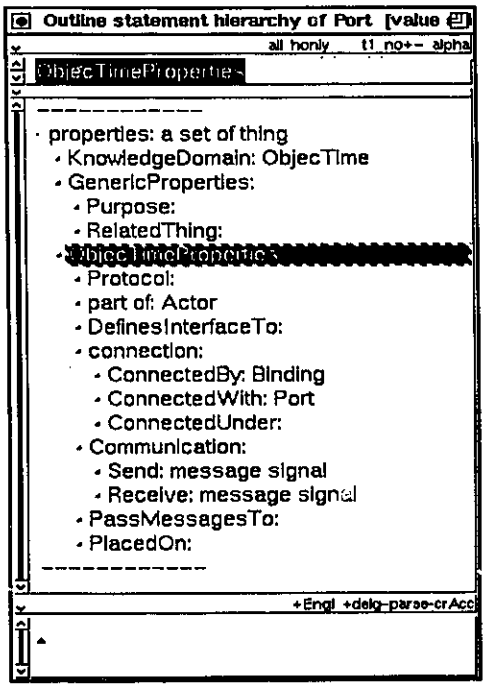


FIGURE 3.4 *Properties of concept port as defined in a CODE knowledge base*

Similar to actors, ports can also be classified into a number of subconcepts (Figure 3.2). An *end port* is a port which can pass messages to the actor's behavior, and it can be directly accessed by the actor's FSM. It can be classified into *internal end ports* and *external end ports*. The former are ports which are used to connect the behavior component to an internal component actor, and are not visible from the outside of the actor. The later are ports which are visible on the outside of an actor and which are connected directly to the behavior. It is also a kind of interface port since it connects two actors. Another kind of interface is a *relay port*, which also appears on the interface of the actor and must be connected to a component actor. Messages coming through a relay port are not visible to the behavior of the actor, instead, they are relayed or funneled directly to the component actor.

Besides the end port and interface port, another category of ports is the *replicated port*. As replicated actors, the concept of replicated ports is provided for designs in which multiple references to a common port class is required.

A port defines a *protocol*, which consists of a set of valid message types. Only messages with these types are allowed to pass through the port which defines the protocol.The common specification for all port instances that respond to the same protocol forms a protocol class, which is organized in an inheritance hierarchy in ObjecTime.

Messages

Messages represent information that is exchanged between actors along bindings and packaged into discrete units. They are the only means of communication between actors. A message is defined by three components: signal, priority, and data (Figure 3.5). The behavior of an actor can only be deduced from the outside by observing the flow of messages on its ports.

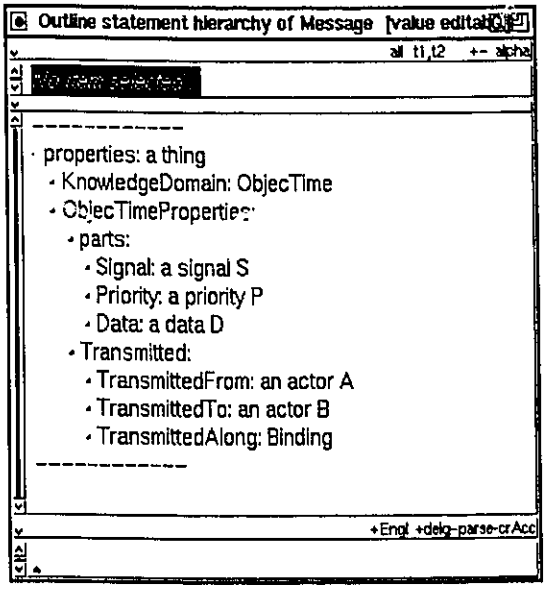


FIGURE 3.5 *Properties of concept message*

Behavior of Actors

As actors are structure entities, they provide a logical container for behavior. The specification of this behavior defines how an actor reacts to external events that take the form of message arrivals. Although in principle ROOM does not prevent any particular formalism from being used to specify behavior, it currently uses an extended FSM formalism to specify the behavior of actors, since this formalism has widespread acceptance in the real-time system application domain. More detailed information about this will be discussed in the next chapter.

Service Access Point (SAP) and Service Provision Point (SPP)

SAP and SPP are system level concepts. ROOM provides a layered approach for architecture. A layer provides a set of services to the entities in the layer above. These entities may provide higher-level services to the next layer above. SAP and SPP

are linkage points between layers and provide means for inter-layer communication. A SAP is the linkage point to the upper layer, while a SPP is the linkage point to the lower layer. Each SAP is mated to a corresponding SPP in the layer below, and they must have matching service protocols to be joined properly. Figure 3.6 provides a short list of properties for SAPs in a CODE knowledge base.

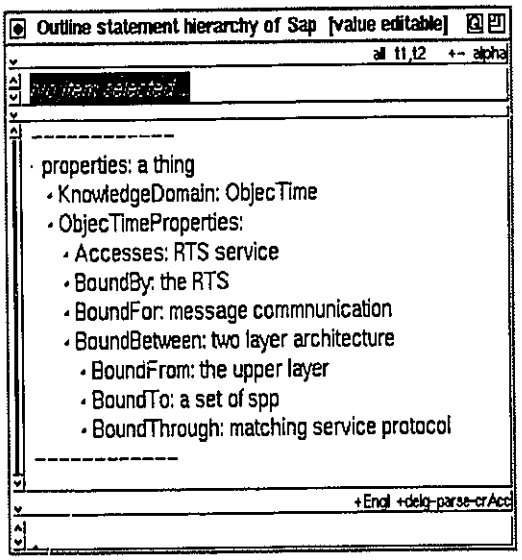


FIGURE 3.6 *Properties of SAP*

ObjecTime Classes

ObjecTime, the supporting tool set for ROOM, organizes designs into three types of objects that can be captured within inheritance hierarchies: Actor class, Protocol class, and Data class. The first two are at the concurrency level, while the last one is at the detail level. These classes are arranged in hierarchies such that they inherit the attributes of their parent classes, and pass on their combined attributes to their children. Actor classes capture the design frameworks, i.e. architecture, and the

complex concurrent behaviors. The Protocol classes capture protocol specifications and support easy generation of protocol variants. Data classes provide a template for the construction of data incarnations used in transition code, state entry / exit code, or as the data for messages.

3.3.3 Main System Features

As we have introduced some of the basic concepts of the ROOM methodology and its supporting tool set ObjecTime, we now describe their basic features in this section.

- *Object-orientation for real-time system development:* As applications in the real-time domain are most naturally expressed as networks of co-operating encapsulating components ([OBJ92]), it is natural to design these application systems using an object-oriented methodology. ROOM addresses this need.

- *Reflects industry needs and experience:* ROOM and ObjecTime is developed by engineers with extensive real-time distributed telecommunication systems development experience. Therefore, the needs of real-time application domains are very well reflected.

- *Supports a highly-iterative computer-aided construction process:* The ROOM methodology supports an iterative and overlapped development process, i.e. activities are not serialized, but overlapped.

- *Supports early design*: As ROOM supports an overlapped development process, the design process starts when the discovery process starts. Such an approach enables the early exploration of risky alternatives.

- *Supports different levels of abstraction:* As ROOM provides two complementary paradigms as its conceptual framework, it has different concepts and formalisms for different levels of abstraction.

- *Supports design clarity and communication:* The methodology and its supporting tool set support rapid visual communication of specification and designs among development team members, and also enable the individual designer to have a clear and concise view of the design.

- *Applies to a broad application spectrum*: The methodology and the tool set can be applied to a broad application spectrum, ranging from networking modeling, high-level architecture, to detailed software design.
- *Supports early execution*: One of the advantages of using ObjecTime is that a design can be executed at an early stage. This feature allows the designer to find out the design mistakes sooner.
- *Potential for incorporating a programming language at the detailed level*: Currently ObjecTime supports C++ and a Smalltalk-like Rapid Prototyping Language (RPL).
- *Supports rapid prototyping and integration*: The methodology and tool set provide rapid iteration and validation of design essence at various levels of abstraction. It promotes fast investigation of promising design alternatives, and uncovers specification deficiencies at an earlier stage.
- *Hypertext style browsing*: ObjecTime provides a hypertext style browsing system which enables the user to open a browser on any item just by pointing the mouse on the item and click on it.
- Automatic code generation: ObjecTime is also a code generator which can produce an executable C++ or RPL code based on the design, i.e. the implementation of the design is automated.
- *On-line help documentation:* ObjecTime provides an on-line help system which uses a hypertext style natural language.

3.3.4 Shortcomings

- *Lack of machine support for requirements analysis*: As a modeling methodology, ROOM provides a conceptual framework for software design. However, it does not provide support for capturing requirements from the application domain and mapping these requirements into the ROOM conceptual framework.
- *Lack of knowledge-based support*: ROOM and ObjecTime does not provide knowledge-based support for capturing domain knowledge, design knowledge, and implementation knowledge in the application domain. It depends on the user's experience in the application domain to carry out the

design task, and the associated knowledge in the above mentioned three aspects can be either lost or buried in the design. As a result, it is difficult to perform efficiently use this knowledge.

- *Lack of systematic documentation capacity for the design:* ObjecTime provides a hypertext style documenting facility for each design component. However, this facility just provides a text editor and therefore only supports unanalyzed natural language comments. The only knowledge operations such a facility can provide are storage, retrieval, and limited formatting capacity. Although the comments for each design component are associated with that component, ObjecTime does not provide any inference capacity to perform more sophisticated knowledge operation such as inheritance, consistency checking, terminology checking, or comparison.

- *Combining design with implementation details*: As a code generator, ObjecTime requires the designer to write code on the transitions of ROOMCharts in order to complete the specification of an actor's behavior. Hence consideration of implementation details are required early in design stage. Meanwhile, a particular implementation language (either C++ or RPL) has to be selected to code the transitions, which means that the design is not independent from the implementation.

CHAPTER 4

An Ontology for Behavior-related Concepts

This chapter will illustrate the use of CODE as an assistant to performing a conceptually-oriented analysis leading to an ontology for behavior-related concepts. The finite state machine formalism, a powerful formalism widely used to describe the behavior of entities, will be reviewed and several extensions of the formalism will be examined.

4.1 Introduction

It has been widely recognized that there exists a major problem in the specification and design of large and complex reactive systems, such as telecommunication systems, computer operating systems etc., which is rooted in the difficulty of describing reactive behavior both in a clear and simple ways and in formal and rigorous ways. Since the natural behavior of a reactive system can be viewed as the set of allowed sequences of input and outputs, conditions, and actions ([HARE87]), the FSM formalism is one of the natural techniques widely used to describe the dynamic behavior of a complex system. The behavior of such systems can be described in a general form of state transition rules: when event *a* occurs in state A, if condition C is true at the time, the system moves into state B.

Although the FSM formalism has been widely used in software engineering practice, it seems that there is no common agreement on several key concepts related to the formalism, especially the concepts of "state" and "event". For example, there exists a number of variants of Harel's statechart formalism ([HARE87]), which is an extension of conventional FSM formalism. The basic concepts of state and event in these variants are somewhat different, but this is not immediately apparent. As another example, the Object Modeling Technique (OMT) employs an extension of statecharts to model the dynamic behavior of a system ([RUMB91]), and recognizes the association between states and activities which implies that some "states" may have some notion of activity associated with them. We will call such states "dynamic". In contrast with the recognition of a dynamic nature of some states, The Real-time Object-Oriented Methodology (ROOM), which also employs an extended statechart formalism, considers "state" as a static situation ([OBJ92]). Furthermore, OMT defines an "event" as something that happens at a point of time, therefore, has no duration. However, it also defines event as one-way transmission of information from one object to another object, which suggests that an event must have a duration. [OBJ92] explicitly states that an event has an (user specified) duration.

These are just few examples of the confusion surrounding behavior-related concepts when using FSM formalism. It seems therefore that there is a need for a conceptual analysis of these concepts, so that a clearer ontology may be established. In this chapter, we use CODE as an analytical assistant to conduct such an analysis. First, we review some basic concepts of the FSM formalism and discuss some of the existing confusion in the formalism. Next, we examine several extensions of the formalism. A section then is denoted to a conceptual analysis of some behavior-related concepts.

4.2 The Finite State Machine Formalism

Finite state machines are a basic computer science concept that is described in any standard text on automata theory. They are often described as a recognizer or generator of formal languages. This formalism is widely used in structured as well as object-oriented analysis and design methodologies for requirements analysis and specifications to describe and model the dynamic behavior of a system. Conventionally, a finite state machine is a hypothetical machine that can be in only one of a given number of states at any specific time. It responds to an input, and possibly generates an output and changes to the next state. Both the output and the next state are purely functions of the current state and the input ([DAV90]). Typically, a finite state machine consists of a finite set of states, a finite set of inputs and outputs, and a transition function ([GHE91]). It is suitable for describing systems that can be in a finite set of states and that can go from one state into another as a consequence of some events.

The key concepts in the FSM formalism are *state, event*, and *transition.* Despite the wide application of the formalism, in many cases it is applied without a clear and explicit definition of these key concepts, and considerable confusion may result. In the following paragraphs, an effort to explore the meaning of these concepts are made. We next show how several authors view the notion of "state".

According to [RUMB91], a state is an abstraction of the attribute values and links of an object. It captures the values of attributes according to properties that affect the gross behavior of the object. A state specifies the response of the object to input events. It corresponds to the interval between two events, and has a time duration.

The following is a definition of state, given by [BOOC91].

> *The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.*

This definition, according to the available material currently, is so far the best and the most clear definition of the term state. But it leaves much to be desired. Further and detailed analysis about this concept will be carried out in a latter section.

Event is probably one of the most unclear concepts in the FSM formalism. There is no doubt that an event is a stimulus from one object to another. One event may precede or follow another event, or two events may be concurrent. In [RUMB91], event is defined as something that happens at a point of time, therefore, it has no duration, or at least the duration is insignificant compared to the granularity of the time scale of a given abstraction. In [OBJ92], event is defined in terms of a transition as a set of signals, a set of end ports, together with an associated guard condition, and it is the arrival of a message. In many other methodologies and systems, the term event is thought of as widely used without a clear definition or discussion. [BOOC91] tells us only that events are "things that happen", but provides no further discussion about this important concept. Similarly, [DAV90] just mentions that events are stimuli without any further description and discussion of the concept. In a well-know KMS, know as Botany ([PORT88]), the terms event and subevent are used to define other concepts in a knowledge base to capture knowledge of the first year university Botany course, but event itself is not defined in the knowledge base at all.

Transition is yet another important concept associated with the FSM formalism. A *transition* is a change of state caused by an event. In other words, a transition can be described as a triple: <current state, event, next state>, which means that a transition is triggered by an event, and leads the object from the current state into the next state. In a state transition diagram, a transition is drawn as a directed arc from the receiving state to the target state. The label on the arc is the name of the event causing the transition in Moore model, and also the action associated with the event in a Mealy model.

Transitions can be further classified into out-going transitions, in-coming transitions, and internal self transitions, according to the relation with the current state. For a current state S, an in-coming transition is a transition which leads the object from another state C into the current state S. An out-going transition is a transition which leads the object from current state S into a next state C, and an internal self transition is a transition which does not lead a state change.

4.3 The Extension of Finite State Machine Formalism

The conventional FSM formalism tend to produce large unstructured diagrams which are complex and difficult to understand, and such a situation is often error prone. To overcome this shortcoming, a significant extension of the basic FSM formalism, *statecharts*, was proposed by [HARE87]. It is the most successful effort to date to structure finite state machines so that combinatorial explosion of states can be avoided. Statecharts are part of a larger development methodology that has been implemented as a commercial product called *Statemate* ([HARE88]). This section provides a brief overview of the statecharts extension, and the extension to the statecharts notion.

4.3.1 Statecharts

[HARE87] presents a broad extension of the conventional FSM formalism and state transition diagrams. The diagrams proposed in this significant work are called statecharts, which employ a contour-based notation for state diagram, and extend conventional state transition diagrams with essentially three elements: hierarchy, concurrency, and broadcast communication. The key concept of these extension is the concept of *super state*, which allows a state to be decomposed into a number of substates, and a state may also be refined into yet another state transition diagram. By such an approach, the extension produces modular, hierarchical, and structured specification.

Hierarchies allow common transitions from states to be clustered. Figure 4.1 shows a simple state transition diagram, in which transition b will leads the system from either A or C into state B. In a statecharts representation, we may group state A and C into a super state D, and cluster the two transitions b in Figure 4.1a into a single transition b as in Figure 4.1b from the boundary of D. The super state D here is an exclusive-OR (XOR), because to be in state D, the system must be in either A or C, but not both. This generalization of state A and C into a super state D is accomplished on the basis of the common property that b transforms the system into state B.

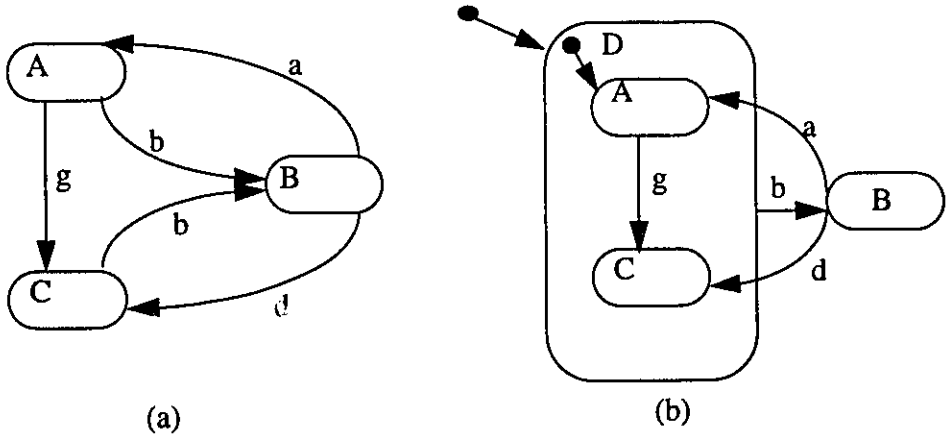


FIGURE 4.1 *Modeling of XOR relationship via statechart clustering (Redrawn from [HARE87])*

Concurrency allows an object to be in more than one state, which is a major extension of the traditional concept of FSM, in which an object can only be in one state. This feature provides a means to model the concurrent behavior of large real-time distributed systems. It can be represented by AND decomposition of states. When an object moves into a state which is AND decomposed into several substates, it must be in all of its AND component substates. The notation used in statecharts is a dotted line separating the concurrent components. Consider the example in Figure 4.2. The state Y consisting of AND components A and D. Being in state Y means being simultaneously in both state A (B or C) and D (E or F or G). Suppose that state B and F are the initial states, which means that when the system moves into state Y, it is in both states B and F. It is interesting to note that the event a will lead the system into state C and G simultaneously. This provides a kind of synchronization. On the other hand, when the system is in state C and G, the event b will only lead the system into state B and state G remains unchanged. This illustrates a certain kind of independence, since the event only affects the A component.

It is important to note that in the statechart notation, every state transition diagram has a start state, which is its default state, pointed out by an arrow starting from a dot. For example, in Figure 4.1, state D is the default state among D and B, while A is the default state among A and C. In Figure 4.2, state B is the default state for A, and F is the default state for D. This requires that when the system moves into state Y, it must be in state B and F simultaneously, unless otherwise specified.

Statecharts have many other features, which are not reviewed here. Only the portion that influences the conceptual analysis in a latter section is discussed here.

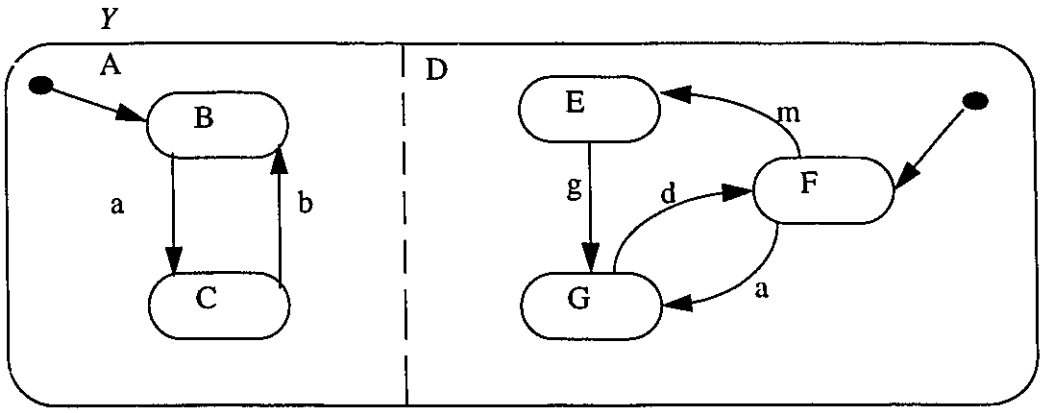


FIGURE 4.2 *AND composition in statecharts (Redrawn after [HARE87])*

4.3.2 Object Modeling Technique

[RUMB91] provides a well developed technique, known as Object Modeling Technique (OMT), for object-oriented modeling and design. It employs Harel's statecharts formalism to model the dynamic and control aspects of a system, and provides a lengthy discussion about state and event. [RUMB91] defines state as an abstraction of the attributes and links of an object, which specifies the response of the object to input events. The response of an object to an event may include an action or a change of state by the object. It is one of the important concept in OMT that state is associated with activity. Furthermore, it recognizes that some states may be associated with activities which take time to complete, such as flying from Ottawa to Toronto, while other states may often be associated with continuous (repeated) activities, such as ringing of a telephone.

As stated earlier, [RUMB91] defines event as something that happens at a point in time, therefore has no duration, or at least the duration is insignificant compared to the time scale of a given abstraction. Among all the features recognized in this work, the authors introduced the concept of grouping events into *event class* and give each event class a name to indicate common structure and behavior, although every event is a unique occurrence. This structure is hierarchical, and most event classes have attributes indicating the information they convey.

One of the differences between the state transition diagram in OMT and statecharts is that every state diagram in statecharts has a default or starting state, while OMT's state diagrams can also represent continuous loops without the default or starting state. For example, the state diagram for the phone line as abstracted in figure Figure 4.3 is a continuous loop, and in describing ordinary usage of the phone, we do not know or care how the loop started. The second type of state diagram in OMT is the one-shot life cycle representation, which represent objects with finite lives, and has initial and final states. The initial state is entered on the creation of the object, while entering the final state implies destruction of the object. As in statecharts, OMT employs nested state diagram, as well as the notation to show concurrency.

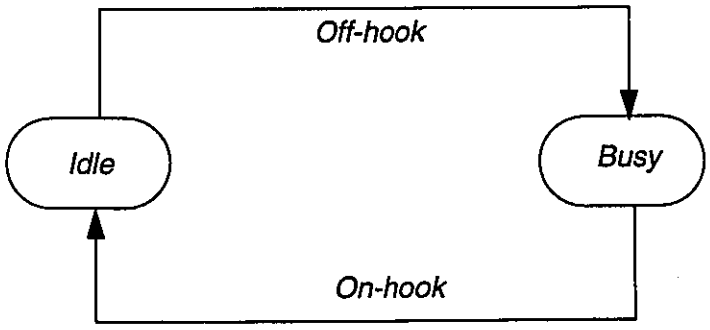


FIGURE 4.3 *A state diagram without starting state*

4.3.3 ROOMcharts

[SELI93] and [SELI94] presents an object-oriented variation of the statecharts formalism for distributed real-time systems, known as ROOMcharts. It is part of the ROOM methodology. ROOM recognizes that the statecharts formalism has some basic concepts which are difficult to implement for real-time systems, and integrates statecharts with the object paradigm. Therefore, ROOMcharts are developed to describe the behavioral aspect of a real-time object-oriented system.

ROOMcharts inherit the concept of super state, i.e. they allow a state to be decomposed into substates, and a contour-based structure is employed. This allows complex behavior to be defined gradually as a series of nested behavior patterns. A ROOMchart state is composed of the following optional attributes: a set of extended state variables, an entry action, an exit action, a set of (sub)states, and a set of transitions ([SELI93]). Extended state variables are used by a FSM to maintain auxiliary information that it needs to sustain between events, and can be manipulated by the code inside transitions and entry and exit actions. Entry actions are optional code segments that are automatically invoked whenever a transition enters the state, no matter which transition was taken into the state. Exit actions are similar to entry actions, except that they are activated whenever a transition is taken out of the state. A transition can be triggered by the arrival of a message (an event), and can optionally have a code segment associated with it which captures the detail-level behavior associated with event handling. ROOMcharts propose a new concept, known as *choice point*, beyond statecharts and conventional FSM formalism.A choice point is used to split a transition into multiple transition segments according to the truth value of a boolean expression associated with the choice point. Another important concept of ROOMcharts is event. Compared with the definition of event in other studies, an event in ROOMcharts is defined as the arrival of a message at some end port, and consists of a port, a signal, a priority, and an optional data item. It triggers a transition, which performs the event processing and leads the actor from its current state into the next state.

Figure 4.4 shows an example of a ROOMchart. A typical ROOMchart, i.e. a FSM description of the actor behavior, consists of the following attributes: a set of end ports, a set of service access points, a set of extended state variables, a set of internal functions, an initial point, a set of states, and a set of transitions. The initial point in ROOMcharts is similar to the start point in statecharts, which points by a transition to the default state unless otherwise specified. Every ROOMchart has an initial point. The choice point, C in Figure 4.4, splits transition e1 into two segments, pointing to state S2 and S3 respectively. Internal functions contain common code sequences that can be shared by event handling code.

In summary, ROOMcharts are a variant of statecharts with emphasis on the object-oriented paradigm, and fully encapsulated within actors which have explicit multiple interfaces. Statecharts have no such interfaces at all. Perhaps one of the most important distinctions between ROOMcharts and statecharts is the lack of concurrent states in the same object in ROOMcharts. ROOMcharts employ concurrent communicating actors rather than concurrent states, so that the communication relationship can be explicit.

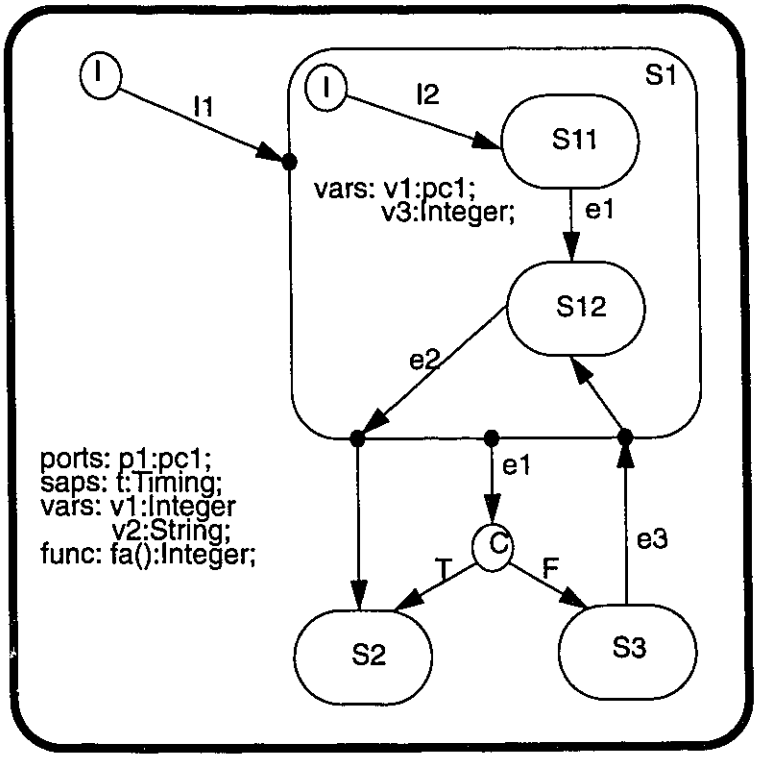


FIGURE 4.4 *An example ROOMchart (Redrawn from [SEL93] with permission of the author)*

4.4 The Concept of Situation

In this section we describe an ontology of some behavior-related concepts we have defined in a CODE knowledge base (Figure 4.5). Although all knowledge representation systems have their own ontology, it seems that many of them include some notion of "situation" in their ontology. Situation is a very abstract concept. The major properties we define at this abstraction level are summarized in Figure 4.6.

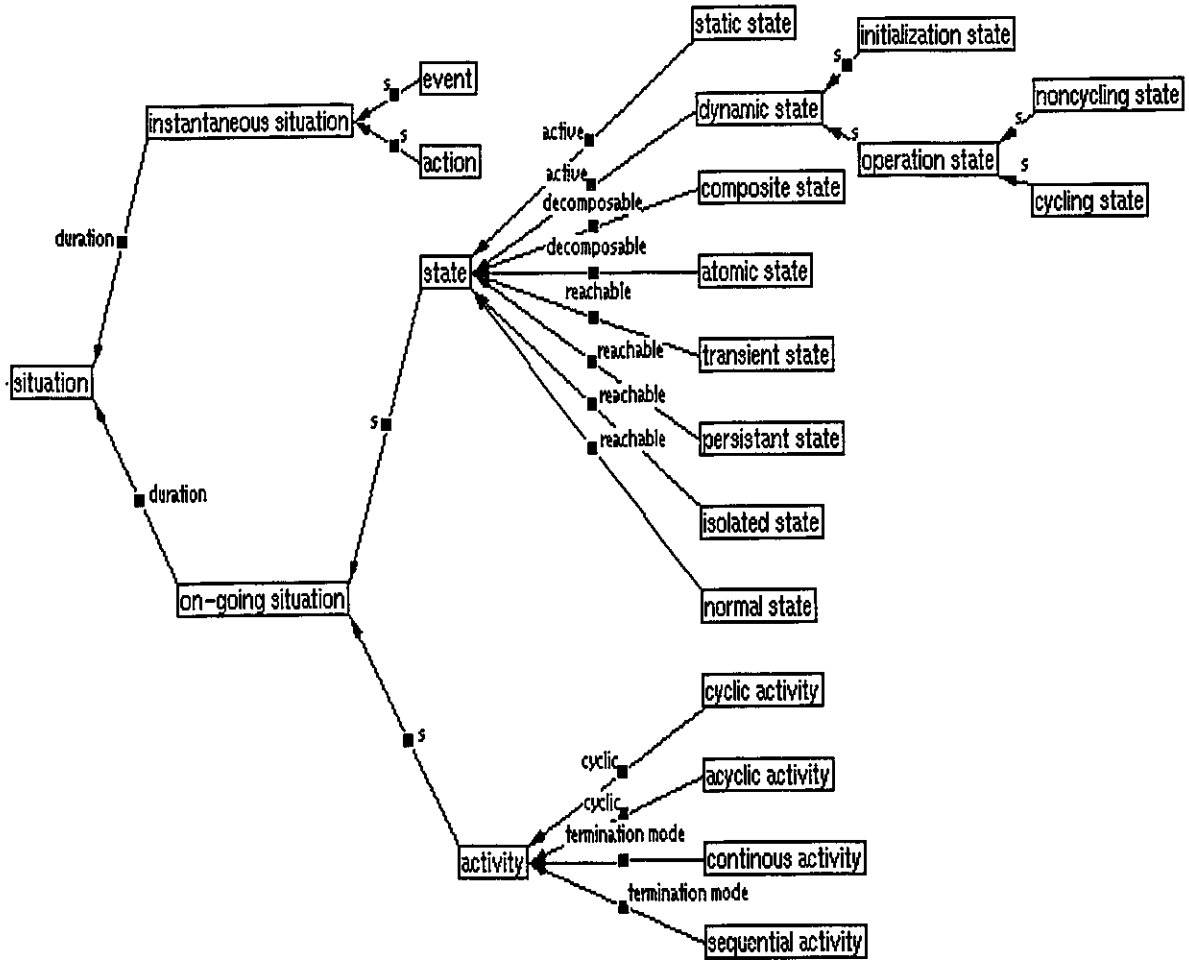


FIGURE 4.5 *The ontology of behavior-related concepts. The labels, e.g. duration, indicate the basis of the distinction.*

Situations may be classified into two disjoint groups, as indicated in Figure 4.5. The first group is instantaneous situation which includes event and action. The second group is occurrence or on-going situation which includes state and activity. The latter has duration.

4.4.1 Instantaneous Situation

An instantaneous situation has no duration, or the duration is insignificant in comparison with the time scale of a given abstraction. In this subsection, we discuss the properties of event and action.

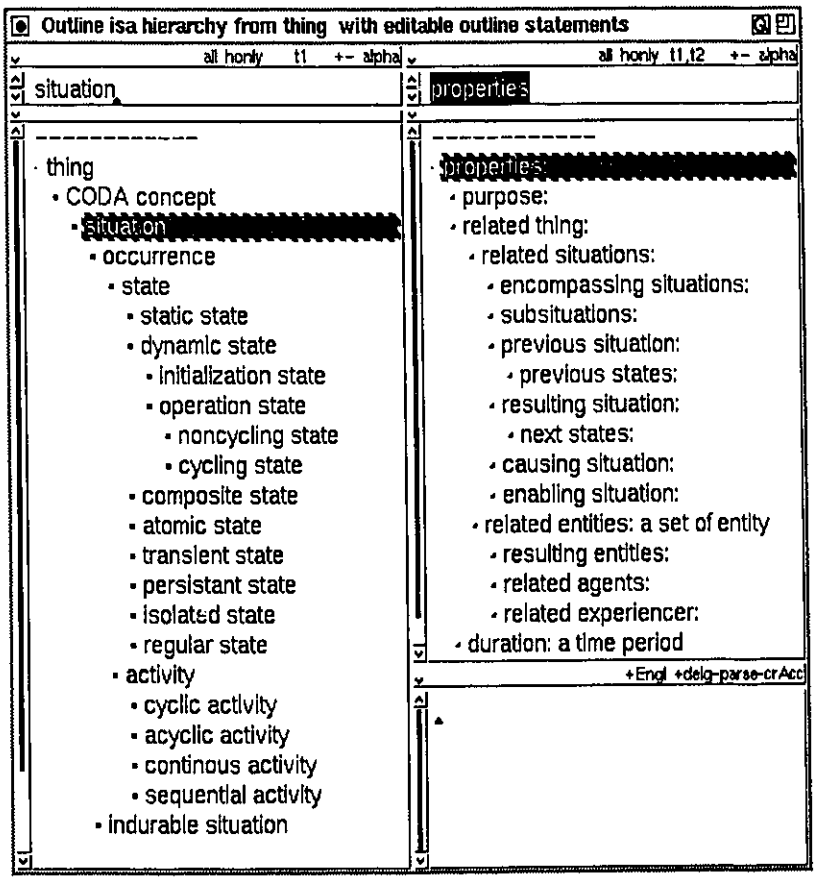


FIGURE 4.6 *Proposed properties of situation (right pane)*

4.4.1.1 Event

The best description of possible properties of event is that given in [RUMB91]. It is commonly agreed in the literature that an event is something that happens. Furthermore, we adopt the notion from [RUMB91] that an event may have no duration or at least its duration is insignificant in comparison with the time scale abstracted by a FSM. Also, an event is associated with an action, and triggers a transition which is a change of state of the object. An event may have some related events, a set of preceding events and a set of following events. It also has a set of related states, as it separates two states. An event has an enabling situation which is an on-going situation, and also may have a set of resulting entities.

4.4.1.2 Action

Similar to event, an action is an instantaneous operation, and is associated with an event. It represents an operation whose duration is insignificant compared to the resolution of the state transition representation, therefore, may be considered as having no duration. It differs from an event by having related patients which are entities this action will act on, and agents which are entities performing the action.

4.4.2 On-going Situation (Occurrence)

In comparison with an instantaneous situation, *on-going situation* has a life time or duration. On-going situation may be further classified into activity and state. In this subsection, we describe the general properties of activity and state.

4.4.2.1 Activity

The properties of activity may be summarized in two aspects. First, an activity is an operation that takes time to complete, therefore it has duration. Second, it is associated with and controlled by a state. Activity may be further classified into two dimensions, each consisting of two subtypes. The first dimension is determined by the repetition nature of some activities, which includes cyclic activity and acyclic activity.

A *cyclic activity* includes a number of repeated subactivities. Ringing a telephone bell is a cyclic activity, because the bell rings and stops in a infinite loop until one picks up the phone or the caller hangs up. Walking is another example of a cyclic activity. A robot transferring a number of boxes from one room to another is also a cyclic activity. An *acyclic activity* can not be decomposed into repeating subactivities. Flying of an airplane and performing of a ballet are examples of acyclic activity. The second dimension is based upon the termination style. According to this criteria, activity may be classified into continuous activity and sequential activity. A *continuous activity* persists until an event terminates it by causing a transition from the current state to the next state. Displaying a picture on a television screen and ring a telephone are examples of continuous activities. Another kind of activity is *sequential activity*, which progresses until completion or until it is interrupted by an event that terminates it prematurely. Examples of this kind of activity include flying of an airplane, performing a ballet, and a robot transfers a number of boxes from one room to another. There is no one to one matching among those two dimensions. An sequential activity may be cyclic activity, such as transferring boxes by a robot from one room to another, or may be acyclic activity, such as flying from Ottawa to Toronto. Similarly, a continuous activity may be a cyclic activity, such as ringing a telephone, or an acyclic activity, such as heating a room by a electrical heater without a thermostat. The differences of these subtypes may be better illustrated using CODE's property comparison matrix feature (Figure 4.7).

4.4.2.2 State

Among those concepts analyzed in this chapter, state is perhaps the most important concept related to the FSM formalism, and a widely and ambiguously used term inside and outside computer science. The definition of state given by [BOOC91] (section 4.2) is the most meaningful one yet in the literature. As a result of a conceptual analysis of the usage of term state in literature, we formulated a property hierarchy associated with state as illustrated in Figure 4.8.

Matrix of statements of (cyclic activity, acyclic activity, continous activity, sequential activity)

	cyclic activity	acyclic activity	continous activit	sequential activi
modifying situation				
terminated by completion	n/a	n/a	n/a	an event E1
terminated by interruption	n/a	n/a	n/a	an event E2
rithmic subactivity	a set of	n/a	n/a	n/a
controlled by	a state	a state	a state	a state
terminating situation	an event	an event	an event	one of (E1 , E2
related thing				
related situations				
encompassing situations				
subsituations				
previous situation				
resulting situation				
causing situation	a set of event	a set of event	a set of event	a set of event
enabling situation				
previous states				
next states				
related entities	a set of entity	a set of entity	a set of entity	a set of entity
resulting entities				
related agents				
related experiencer				
duration	a time period	a time period	a time period	a time period
purpose				
properties				

FIGURE 4.7 *Properties shared by at least one subconcept of activity*

It is commonly agreed in the literature that a state may accept a set of exiting events which trigger out-going transitions, a set of entering events which trigger in-coming transitions, and a set of internal events which trigger internal self transitions. Also, a state may be entered from a certain set of previous states, and may change to a certain set of next states. It responds to an input and generates an output, and has duration. A state must have a related experiencer (that is "in" the state) which is an entity, and may have a set of agents (which "cause" it) which are active entities. A

state is typically associated with an activity. From another view, the reachability of a state may be defined thus: it may be entered by a set of in-coming transitions, exited by a set of out-going transitions, and internally accessed by a set of internal self transitions.

Without loosing the generality of state properties defined above, state can be further classified into three viewpoints. All subconcepts in these three viewpoints inherit the general property of state as defined above. The first viewpoint includes two subconcepts, static state and dynamic state, according to the activity. A *static state* is a state which has no activity, such as the state wait or idle. In contrast with static state, a *dynamic state* is always associated with an activity, and can be further divided into initialization state and operation state. The *initialization state* is the same as the starting point in ROOMcharts and statecharts, and the initial state in OMT. It is entered whenever an object is created, and may be associated with an initialization activity. An *operation state* is a regular dynamic state, and can be further classified into noncyclic state and cyclic state, according to the type of activity it is associated with. Apparently, a noncyclic state is associated with an acyclic activity, and a cyclic state is associated with a cyclic activity.

The second viewpoint classifies state into composite state and atomic state, according to the decomposability. *Composite state* is a synonym of super state. It can always be decomposed into a number of substates, therefore it has parts: a set of states and a set of events. An *atomic state*, as demonstrated by its name, is atomic and can not be decomposed at all. Such a state may be static or dynamic.

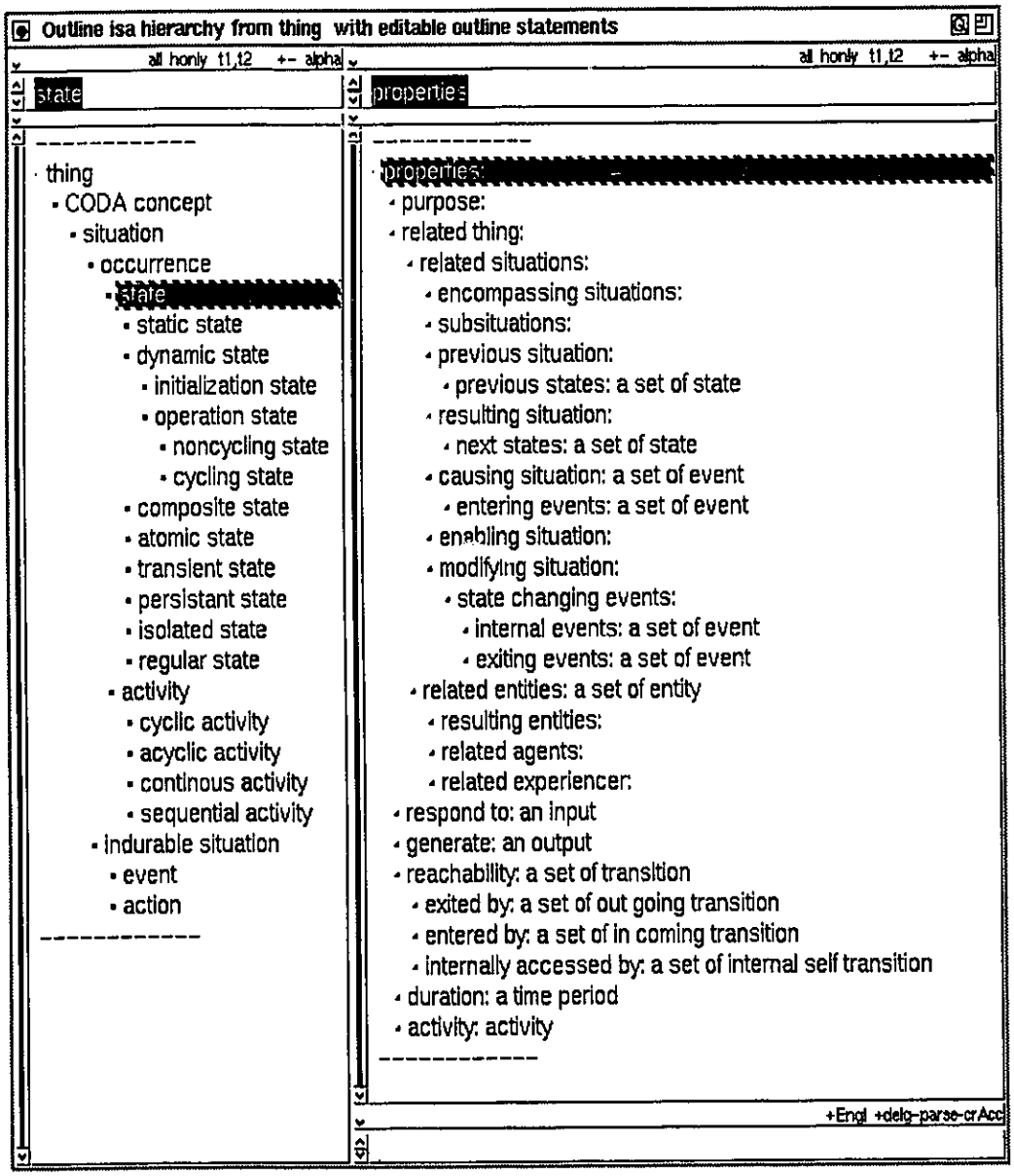


FIGURE 4.8 *The proposed property hierarchy of state (right pane)*

The third viewpoint classifies states into four subconcepts according to the reachability: the transient state, the persistent state, the isolated state, and regular state ([GILL62]). A *transient state* has no in-coming transition, therefore has no previous state and entering event. Such a state can not be recovered whenever it is abandoned. A *persistent state* has no out-going transition, therefore has no exiting events and next states. It can not be abandoned whenever entered. An *isolated state* is a state which has no in-coming and out-going transitions, therefore has no entering and exiting events, and is not related to any other states. It may have a set of internal self transitions. Such a state can not be changed or abandoned. Figure 4.9 illustrates a simple FSM, showing the state classification in accordance with the reachability. The START state is a transient state, and the END state is a persistent state, and state S1 is a regular state which may be entered or exited. State S is an isolated state.

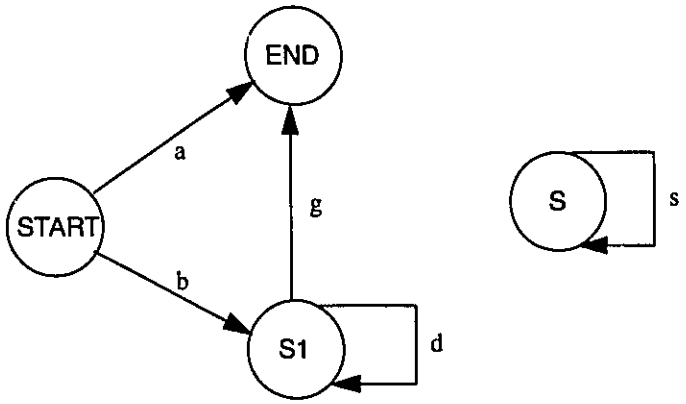


FIGURE 4.9 *Simple FSM showing different states according to reachability*

At this point, it is necessary to carefully analyze which properties make the subconcepts of state different from each other. Figure 4.10 shows the none-nil properties shared by at least one subconcepts of state.

Matrix of statements of (static state, dynamic state, composite state, atomic state)

	static state	dynamic state	composite state	atomic state
substate	n/a	n/a	a set of state	n/a
subevent	n/a	n/a	a set of event	n/a
exiting events	a set of event	a set of event	a set of event	a set of event
entering events	a set of event	a set of event	a set of event	a set of event
internal events	a set of event	a set of event	a set of event	a set of event
respond to	an input	an input	an input	an input
generate	an output	an output	an output	an output
activity	na	activity	activity	activity
reachability	a set of transition	a set of transition	a set of transition	a set of transition
exited by	a set of out going transition	a set of out going transition	a set of out going transition	a set of out going transition
entered by	a set of in coming transition	a set of in coming transition	a set of in coming transition	a set of in coming transition
internally accessed by	a set of internal self transition	a set of internal self transition	a set of internal self transition	a set of internal self transition
causing situation	a set of event	a set of event	a set of event	a set of event
previous states	a set of state	a set of state	a set of state	a set of state
next states	a set of state	a set of state	a set of state	a set of state
related entities	a set of entity	a set of entity	a set of entity	a set of entity
duration	a time period	a time period	a time period	a time period

FIGURE 4.10a *None-nil properties shared by at least one subconcept of state classified by activity and decomposability*

Matrix of statements of (transient state, persistant state, isolated state, regular state)

	transient state	persistant state	isolated state	regular state
exiting events	a set of event	na	na	a set of event
entering events	na	a set of event	na	a set of event
internal events	a set of event	a set of event	a set of event	a set of event
respond to	an input	an input	an input	an input
generate	an output	an output	an output	an output
activity	activity	activity	activity	activity
reachability	a set of transition	a set of transition	a set of transition	a set of transition
exited by	a set of out going transition	na	na	a set of out going transition
entered by	na	a set of in coming transition	na	a set of in coming transition
internally accessed by	a set of internal self transition	a set of internal self transition	a set of internal self transition	a set of internal self transition
causing situation	a set of event	a set of event	a set of event	a set of event
previous states	a set of state	a set of state	a set of state	a set of state
next states	a set of state	a set of state	a set of state	a set of state
related entities	a set of entity	a set of entity	a set of entity	a set of entity
duration	a time period	a time period	a time period	a time period

FIGURE 4.10b *None-nil properties shared by at least one subconcept of state classified by reachability*

Matrix of statements of (initialization state, operation state, noncycling state, cycling state)

	initialization state	operation state	noncycling state	cycling state
exiting events	a set of event	a set of event	a set of event	a set of event
entering events	a set of event	a set of event	a set of event	a set of event
internal events	a set of event	a set of event	a set of event	a set of event
respond to	an input	an input	an input	an input
generate	an output	an output	an output	an output
activity	activity	activity	acyclic activity	cyclic activity
reachability	a set of transition	a set of transition	a set of transition	a set of transition
exited by	out going transition	a set of out going transition	a set of out going transition	a set of out going transition
entered by	a set of in coming transition	a set of in coming transition	a set of in coming transition	a set of in coming transition
internally accessed by	a set of internal self transition	a set of internal self transition	a set of internal self transition	a set of internal self transition
causing situation	a set of event	a set of event	a set of event	a set of event
previous states	a set of state	a set of state	a set of state	a set of state
next states	a set of state	a set of state	a set of state	a set of state
related entities	a set of entity	a set of entity	a set of entity	a set of entity
duration	a time period	a time period	a time period	a time period

FIGURE 4.10c *None-nil properties shared by at least one subconcept of dynamic state*

4.5 Discussion

It is sometimes hard to understand why an activity is associated with a state, other than an event, and why an action is associated with an event rather than state. One may ask a question such as "When we say a flying state is associated with a flying activity, does that means an activity is a state?" or "When a Cancel event and a Cancel action are both labeled on a transition, such as Cancel/Cancel, does that imply that an action is an event or vise versa?". The problem is the confusion about the meaning of these concepts, and sometimes it is a problem of incorrect terminology.

To appreciate the difference between these concepts, let us consider a simple example. Figure 4.11 shows a portion of the specification of the external behavior of a telephone line. Starting from the idle state, the only out-going transition is labeled as off-hook/dial tone, and terminates at dial tone state. There is no activity associated with the idle state, therefore it is a static state. When the system is in this state, the event off-hook, associated with the generating a dial tone action, will lead the system into the dialing state. The dialing state is associated with a sound dial tone activity, therefore is a dynamic state. Here we follow OMT's convention to label an activity with "do:" followed by the activity name. Since an activity is associated with a state, it is labeled inside the state. There are five out-going transitions starting from this state. If a dial valid idle number event occurs, a ring back action is generated, and the system moves into the ring state which is associated with a "do: ring bell" activity, otherwise, if a dial invalid number event or a time-out event occurs, the system moves into the warning state which is associated with a "do: play warning msg" activity. If a dial busy number event occurs, the system moves into the busy state which is associated with a "do: sound busy tone" activity. The behavior of other states are similar and thus do not need detailed explanation. In any of the specified states, if an on-hook event occurs, the system moves into the idle state.

The above example shows the distinction between these often ambiguously used terms. State represents a situation an object is in, and activity represents what is happening when the object is in that situation. Event represents a stimuli which is done to the object, and action represents a response to the stimuli. Here we do not care about

the duration of the response, although in the real world no action is really instantaneous. As indicated in [RUMB91], if we do care about the duration, then we should model that particular response as an activity and associate it with the state, rather than model it as an action.

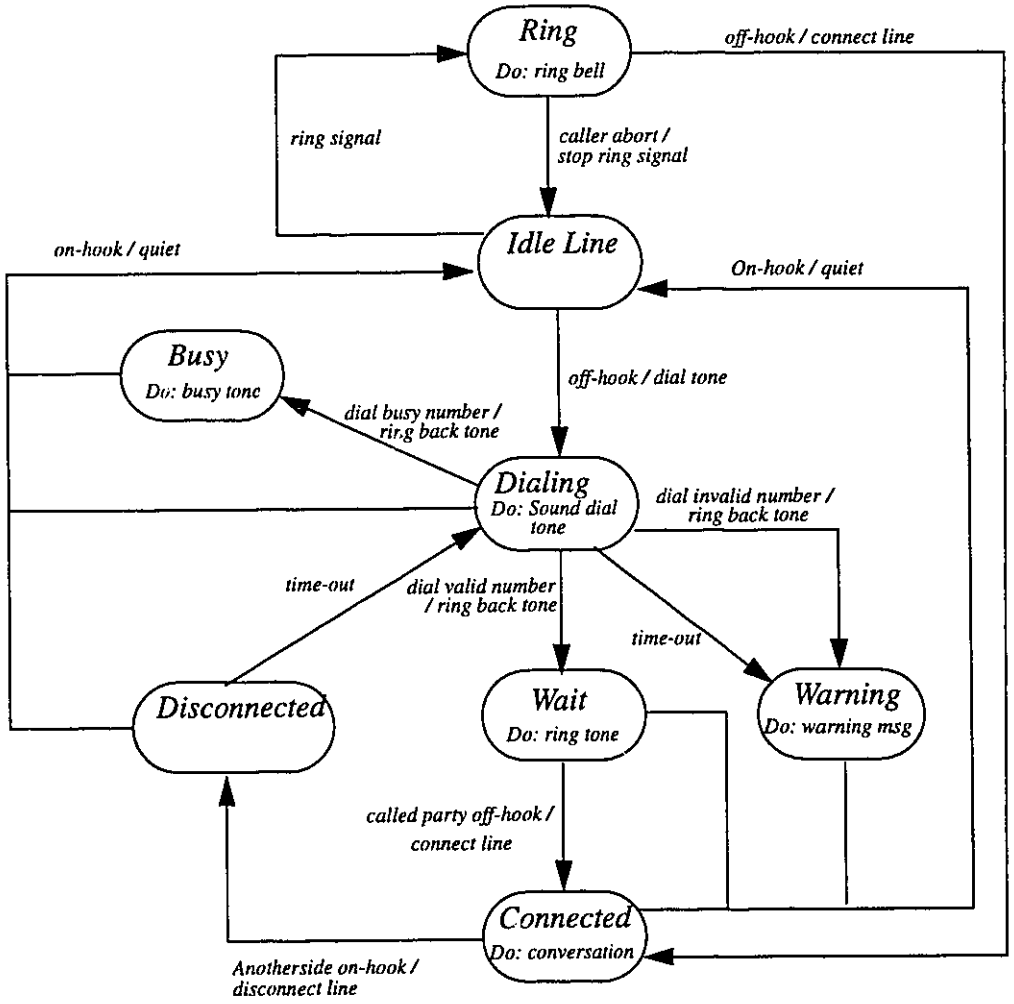


FIGURE 4.11 *A state transition diagram for a telephone line (Do: indicates activities)*

4.6 Summary

In this chapter we have established an ontology for some behavior-related concepts, and discussed the basic characteristics of the FSM formalism and several well established extensions. The FSM formalism is a widely used, well developed mechanism to specify the external behavior of a system. Beyond the conventional formalism, several extensions and variants have been developed and supported by computerized tool sets. The most significant one is the statecharts extension, which expands the formalism in three aspects: hierarchy, concurrency, and broadcast communication. A variant of statecharts, ROOMcharts, emphasizes the object-oriented paradigm, and excludes concurrent states. Instead, it models concurrency by concurrent communicating actors.

With the assistance of CODE, we analyzed several key concepts related to behavior, i.e. the situation ontology, and provided a hierarchical classification of those terms and concepts. A knowledge base was built to clarify the confusion about these concepts in the literature, and to enable the associated properties of these concepts to be clearly defined and easily retrieved whenever needed.

With this preparation, we are now in a much better position to proceed to our main task: developing an ObjecTime design from application knowledge captured in a CODE knowledge base, which will be conducted in the next chapter.

CHAPTER 5

Management of Requirements Analysis and Design Knowledge in CODE

This chapter provides a detailed illustration of how a KMS like CODE may be used to help the software designer acquire application domain knowledge as well as methodology domain knowledge, how to conduct a design using the knowledge acquired during the requirements analysis, and how to use the knowledge base derived during the analysis and design process as an on-line active knowledge resource. An example, the requirements analysis and design of a simple telephone system will be used to illustrate how a conceptually-oriented approach and the knowledge management mechanisms in CODE can help the software designer to build quality software more easily and efficiently.

5.1 Introduction

Software development usually begins with an attempt to recognize and understand the user's requirements. Based upon the analysis and specification of these requirements, a design of the system required can be conducted. The design can then be implemented into a final product. Ideally, the final product should satisfy the original requirements.

During the requirements acquisition process, users and system analysts communicate with each other through documents and dialogues to acquire the necessary generic and application specific domain knowledge, i.e. to derive a requirements definition. The results of this process, which the design will be based upon, reflect the developer's interpretation of the user's needs. Often these needs may be misunderstood due to various knowledge transferring and management problems, which therefore may cause serious errors in the final product. Obviously, managing the knowledge in this process is critical.

Even though a zero fault requirements specification may have been derived, which in most cases will not be likely, it is still almost inevitable that the designed system may not reflect what the user wanted due to imperfect mapping between the requirements and the design. Hence how to derive a correct design based upon the domain and requirements knowledge is a very critical issue.

Although there are many other knowledge management-related problems during the software life cycle as we indicated in Chapter 1, we only focus here on the knowledge management-related problems one may encounter in the upper stream of the software life cycle, i.e. the requirements analysis and design phases. In this chapter, we will use a telecommunications example to illustrate how conceptually-oriented analysis techniques and CODE can help the system analysts to derive and manage the requirements. Then we will illustrate how to use this knowledge to derive an object-oriented design using ObjecTime.

5.2 The Ontological Framework

As indicated in Chapter 1, we focus on two categories of knowledge involved in a typical software development cycle: knowledge in the application domain and knowledge in the development methodology domain. The former is related to the goals of the software development activity, while the latter is related to the tools and techniques that the software developer will use to achieve the development goals. Managing the knowledge in these two different domains, and integrating the knowledge from these two domains together to carry out a particular software development task, requires a common high level ontological framework. This framework plays a critical role in getting the domain experts and the developers to agree on common concepts and terminology. Among its advantages, it promotes knowledge reuse and provides guidance for defining new concepts, because it presents a taxonomy of existing general concepts with their properties. More specific concepts will inherit the generic properties from these generic concepts.

In this chapter we propose an ontological framework which allows users to integrate concepts of multiple methodologies and multiple applications into one single CODE knowledge base. This approach maximizes the reuse of analysis and design components (Figure 5.1). The ontological framework consists of three major sub-ontologies: the domain-independent ontology or generic ontology, the methodology ontology, and the application ontology. Sometimes these are referred to as viewpoints.

5.2.1 The Domain-Independent Ontology (Generic Ontology)

As indicated in [SKUC93c], a carefully specified generic ontology is an essential basis for any serious knowledge base development, and one cannot attempt to organize large varieties of concepts without having a clear picture of the high level concepts and their properties. The domain-independent ontology in our framework contains those concepts which are independent of any particular application or methodology. i.e. very general concepts believed to be useful in virtually any subject area.

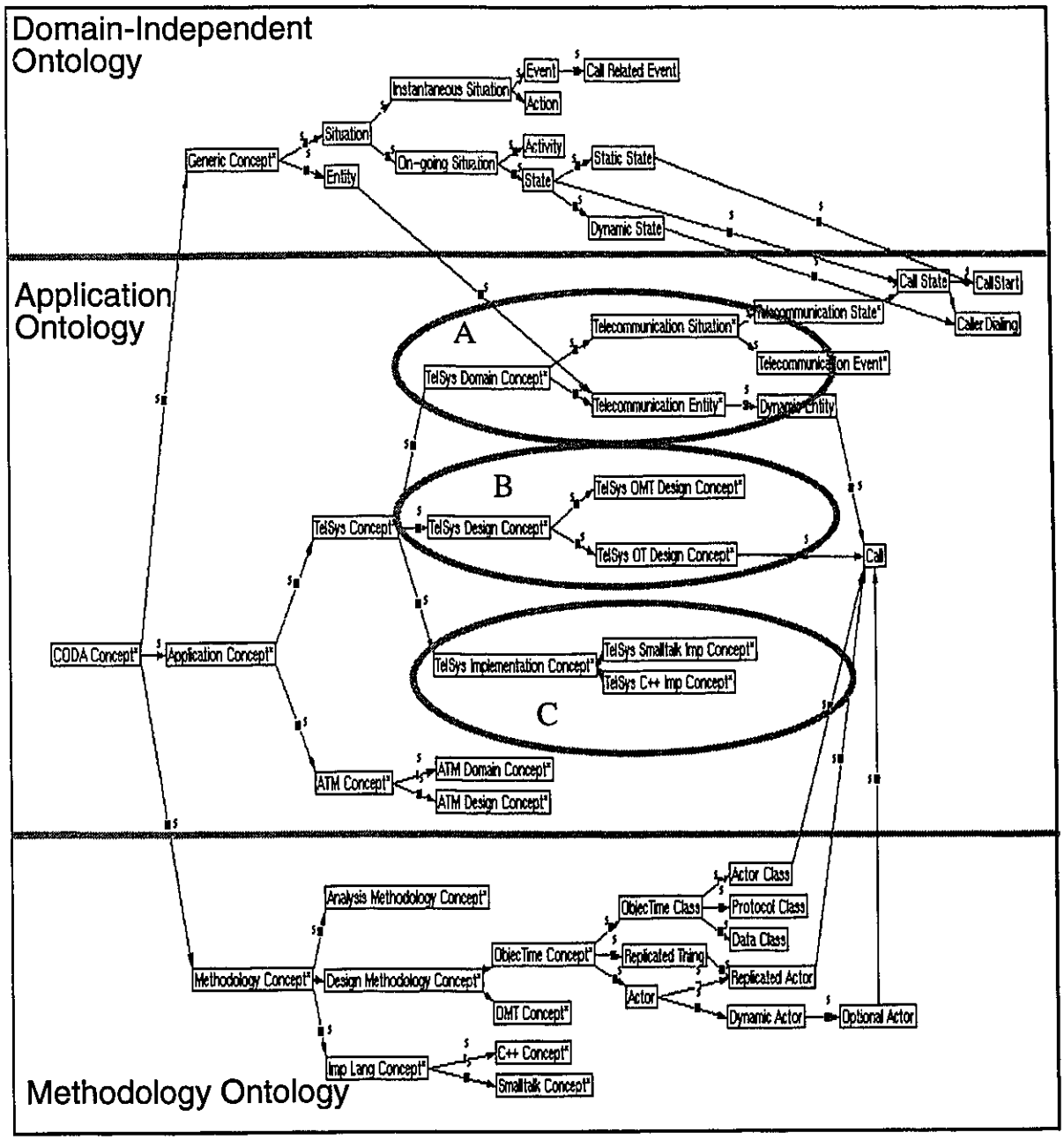


FIGURE 5.1 *Ontological framework for software development described in CODA (Note that not all sub-trees are illustrated). Area A is the sub-tree of Domain concepts; area B is the sub-tree of Design concepts, and area C is the Implementation concepts.*

As the purpose of the work described in this thesis is to show how a KMS like CODE can assist software developers to manage various types of knowledge during software development, we do not intend to provide a complete picture of our vision of the domain-independent ontology beyond any specific domain. Instead, we adopt the high level ontology described in [SKUC93c] with minor modifications of the concept Situation. In other words, the domain-independent ontology in our framework combines the ontology of Situation which we derived in Chapter 4 and part of the top level ontology described in [SKUC93c]. The framework provided here permits future enhancement in this area. Some of the major concepts under this hierarchy are illustrated in the area Domain-Independent Ontology in Figure 5.1. One of the most important generic concepts is *entity*, which has a number of distinctive properties (Figure 5.2).

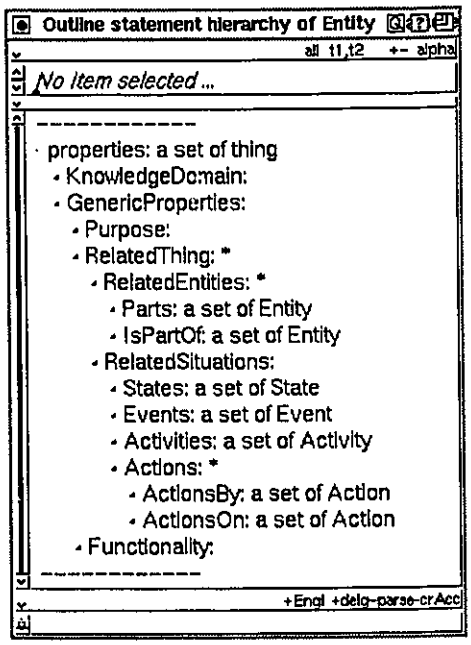


FIGURE 5.2 *Property hierarchy of concept Entity. Note that an asterisk indicates that the corresponding property is a grouping property, i.e. no value applicable.*

The *Purpose* of an entity describes its function, i.e. why it exists. *RelatedThings* describes what other things are closely related to this entity. *RelatedEntities* describes the entities which are related to the current entity. *RelatedSituations* describes what other situations are closely related to this entity. It must be kept in mind that these four properties not only belong to entities, they also belong to any other concepts under the CODA Concept hierarchy, as they are defined at the top concept of our conceptual hierarchy, *CODA Concept*. Other properties illustrated in Figure 5.2 are specific to entities. *Parts* and *IsPartOf* describe a *part-of* and/or *containment relationship* that the entity under description has with other entities. The behavior of the entity can be described by the properties *RelatedSituations* and *Functionality*. We will discuss these properties in more detail later on when we describe certain specific entities.

5.2.2 The Methodology Ontology

The ontological framework we propose provides a platform in which multiple development methodologies can be integrated into the same knowledge base, i.e. the same development environment. We employ a *grouping concept*[1] known as Methodology Concept to indicate that all concepts under this grouping concept are methodology dependent. One may have many methodologies defined in the knowledge base, e.g. an analysis methodology such as Structured Analysis techniques, a design methodology such as ROOM, or an implementation methodology such as Smalltalk or C++. In the area Methodology Ontology in Figure 5.1, we organize the development methodology into three groups: analysis methodology, design methodology, and implementation methodology. Each of these three groups can consist of a number of specific methodologies. These three groups correspond to the three kinds of knowledge involved in a typical software development practice: domain knowledge, design knowledge, and implementation knowledge. Of course, we are not confined to only these three types of methodology. Actually any type of methodology may be integrated into the framework, such as testing strategies etc.

In this thesis, we only define those concepts which are essential to carrying out an ObjecTime design under the ROOM/ObjecTime Concept sub-tree, since we only

1. a grouping concept is a concept used only for grouping other concepts together. It is tagged by an asterisk. This has been commonly used in many CODE knoowledge bases for several years.

use this methodology and tool to perform the design. We will not develop a complete knowledge base about this methodology itself. OMT, Smalltalk and C++ concepts are shown only to illustrate the flexibility of defining multiple methodologies, therefore, no concepts under these sub-trees are actually defined in the knowledge base which we developed.

5.2.3 The Application Ontology

The application concepts are grouped under the grouping concept known as Application Concept, as illustrated in the area Application Ontology of Figure 5.1. Concepts in each application can be organized as a sub-tree, and common concepts between multiple applications need only be defined once with multiple parents. In this chapter, we use a Telephone System example as our main application to illustrate how the requirements analysis can be carried out in CODE and how the knowledge base can be used to assist an ObjecTime design.

While the concepts in the generic and methodology domain ontologies only need to be defined once and then can be reused again and again as a template for other applications, concepts in the application domain ontology will expand when an application grows. All concepts which are necessary for a particular software development task will be defined under this ontology and fit into the other two ontologies.

The domain concepts are separated from the application design concepts and application implementation concepts by organizing them into three groups. For example, the TelSys Domain Concept sub-tree (area A in Figure 5.1) contains domain concepts in our example, while the TelSys Design Concept sub-tree (area B in Figure 5.1) contains design concepts. The implementation concepts should be under TelSys Implementation Concept (area C in Figure 5.1). Since we will only focus on the domain analysis and design, we will not define any implementation concepts in our example, therefore, the sub-tree of TelSys Implementation Concepts in area C of Figure 5.1 is empty.

The domain concepts are concepts of the application domain. They are independent of any design methodology or implementation language, and are commonly used by people who understand the application domain. For example, Call is a typical telephony domain concept that everyone uses every day in modern society, i.e. a telephone call. A design concept is a concept which is defined for a design purpose. It is dependent on a particular design methodology, but should be independent of any implementation languages. Nevertheless, a design concept can also be a domain concept. In this case, Call inherits both domain properties and design properties. Thus it becomes a "mix" of both kinds of properties. As a domain concept, Call has a originator which is a Caller. As a design concept in ObjecTime, Call is defined as an Actor which has properties that only make sense in a design context. For example, a design concept Call in ObjecTime must have a Port. This property, however, is very dependent on the design methodology one uses to carry out the design, and does not make sense for a domain concept. A telephony domain expert may well understand what is a Call, but may not necessarily understand that Call (as an Actor) has ports. We will discuss this in more detail in a later section. CODE permits one to see either or both viewpoints.

To illustrate how a particular lower level concept (i.e. below more general concepts) fits into this ontological framework, we include this concept, Call, in Figure 5.1. This concept is a particular low level concept in the Telephone System application. By multiple parenting, it inherits properties from the generic, methodology, and application domain ontologies. Figure 5.3 illustrates the properties that Call inherits from these three ontologies. Instead of using an outline textual format as in other examples, we use CODE's option to show the property hierarchy in graphical format, which gives a better structural view of a hierarchy. As a design component, the behavior of Call is described by the properties of Call State, which is linked to the concept of Call through property hierarchy. This example also illustrates how these main sub-trees are incorporated.

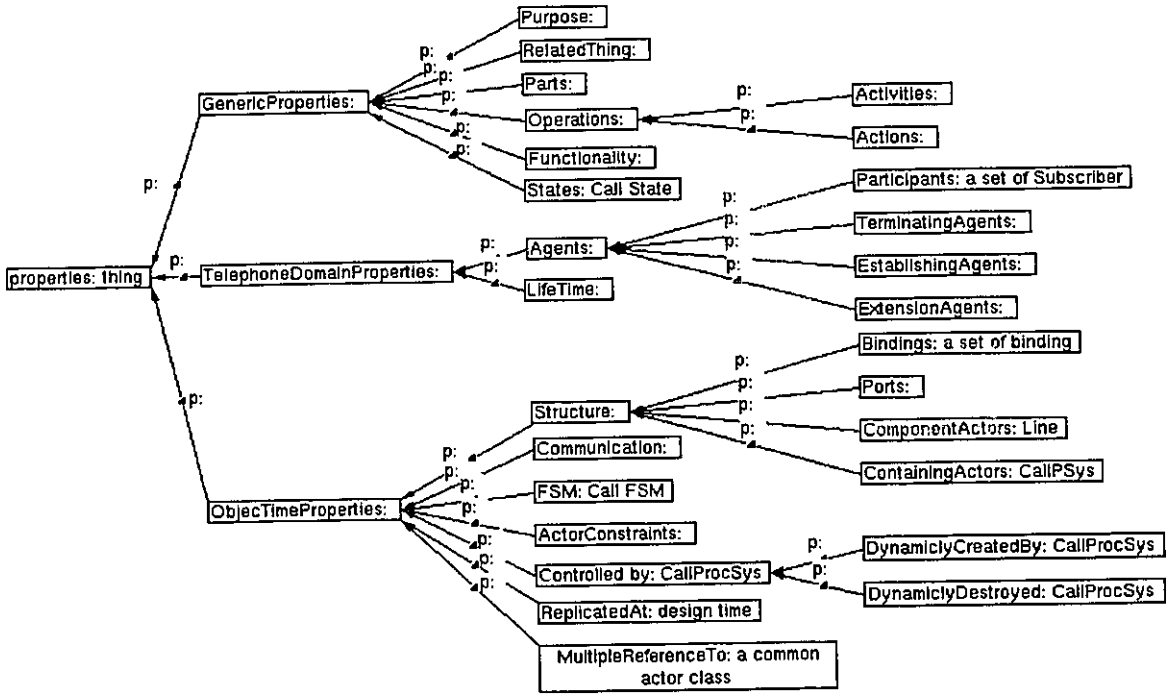


FIGURE 5.3 *Property hierarchy of concept Call in CODA. Note that not all subtrees are fully illustrated*

5.3 Application Knowledge Management: a Telephone System Example

5.3.1 Introduction

Once the basic concepts in the generic and methodology ontologies are defined, we then can use these concepts as templates to guide the application knowledge management, i.e. to carry out the requirements analysis and design. The implementation knowledge acquisition process is similar to that of design knowledge acquisition. The most important aspect of knowledge management is defining the important concepts with their properties. In our conceptual framework, we have

partitioned the properties into three property hierarchies or viewpoints, in correspondence with the three concept ontologies, as some concepts may actually inherit properties from all three ontologies (Figure 5.3). The GenericProperties viewpoint represents properties a concept can inherit from the domain-independent ontology. The methodology-oriented viewpoint (ObjecTimeProperties in this example) represents properties a concept can inherit from the methodology ontology, i.e. the properties specific to the domain in which a certain methodology was employed to carry out a specific development task such as design or implementation. The application-oriented viewpoint (TelephoneDomainProperties in this example) represents the properties specific to the application domain.

In the rest of this section, we will employ a telephone system example to illustrate how CODE can be used as a software development assistant for two major steps in the software development life cycle: the requirements analysis and the design in ObjecTime.

5.3.2 Application Knowledge Encoding Procedures

Based upon the ontological framework we defined in the previous section, we now define the operational procedures of how to encode the application knowledge into this ontological framework. Encoding of knowledge for a specific methodology is the same as encoding of knowledge for a specific application, since in such a situation the methodology under analysis is indeed an application, just like domain analysis of a telephone system.

Typically, the sources of application knowledge come from existing natural language-based documents and brain storming sessions between domain experts and knowledge engineers. If natural language-based documents are available, the Document Processor feature of CODE can be employed to assist the knowledge encoding process (Figure 5.4). The upper window in the figure shows a few sentences from the original natural language documents. The middle window shows that these sentences broken down into fragments suitable for creation of concepts, properties or values. By clicking on the buttons in the bottom window, these sentence fragments shown in the middle window can be added into the associated knowledge base very quickly. With this tool, the person who performs the knowledge encoding may break a

sentence into linguistic parts, and then analyze the relationships between the nouns, their modifiers, and the verbs to decide which should be added into the knowledge base. Normally nouns are good candidates for entities or property values, while verbs and modifiers are good candidates for property names. In general, words in the document can be added into the knowledge base as concepts, properties, values of properties, or informal comments. Instead of using the Document Processor, one may choose to enter the concept directly, especially when natural language documents are not available. Most likely, one would use a combination of the two.

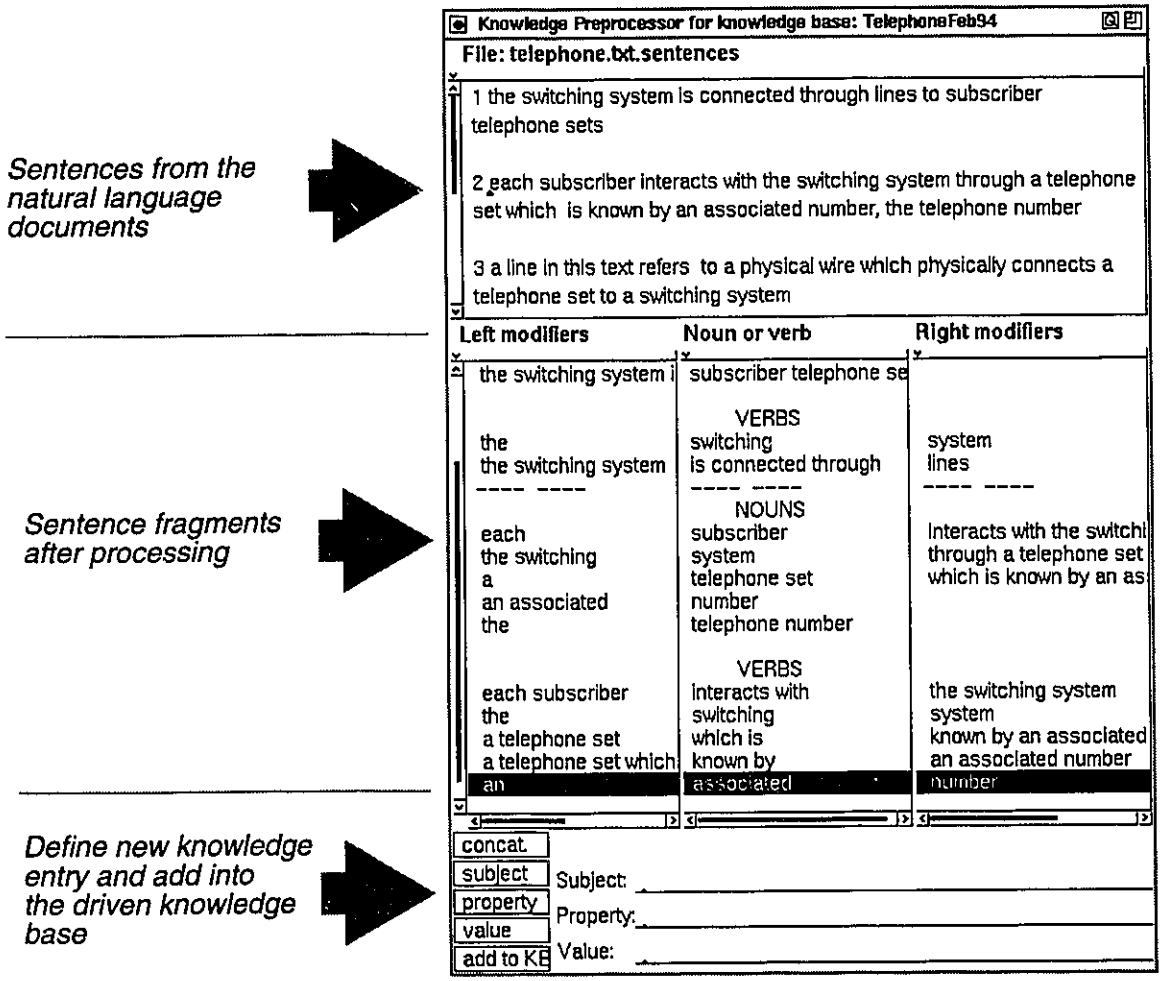


FIGURE 5.4 *CODE's document processor*

Once a concept is chosen for analysis, the first step is to fit it into the existing ontological framework. We start by fitting it into the application ontology, i.e. finding a suitable super concept in the application ontology. Then we try to determine where it would fit in the domain-independent ontology, i.e. finding a suitable super concept in the domain-independent ontology.

The following rules apply in general during the conceptual analysis and requirements acquisition phase:

1. All leaf nodes in the application domain ontology must have at least one node under the domain-independent ontology as co-parent.
2. If an ancestor of a leaf node is fitted into a particular position in the domain-independent ontology, and this leaf node has no new properties defined under the GenericProperties hierarchy and no new value is added to the inherited properties under the same property hierarchy, then rule 1 should be overridden, i.e. this leaf node should not be fitted into the same sub-tree of the domain-independent ontology.
3. None-leaf nodes in the application domain ontology may have one or more nodes under the domain-independent ontology as their co-parent, only when new properties need to be defined under the GenericProperties hierarchy, or a new value of at least one property under the GenericProperties hierarchy needs to be defined.
4. Grouping concepts, which typically have no new property defined and no new value added to the inherited properties, should not be fitted into the domain-independent ontology. This will minimize the overhead created by the multiple parenting.
5. Grouping concepts which can fit into the domain-independent ontology as a leaf node should override rule 4 and fit into the hierarchy, because all their children no longer need to fit into the same place.
6. Concepts which should fit into the generic concept hierarchy must be placed at the lowest possible level of the generic concept hierarchy because the lower that level is, the more properties can be inherited.

During the design phase, the following rules should be applied:

1. Select design concepts from the domain concepts according to the requirements. As the application domain ontology often contains more concepts than needed for the design, selection of the design concepts

should follow the user's requirements, i.e. only those concepts which are necessary for the design should be chosen as design concepts.

2. Each design concept must fit into the methodology domain ontology, i.e. must have at least one methodology domain concept as its co-parent.
3. Except the grouping concept rules, all other rules for the analysis phase should be also applied during the design phase. The difference is to fit the design concept into the methodology domain ontology, not the domain-independent ontology.

5.3.3 The Requirements Analysis: Capturing Domain Knowledge

Software requirements are usually confined to a particular application domain, hence acquiring requirements is actually a part of capturing domain knowledge. In our model, domain knowledge is stored in a CODE knowledge base. The process may begin with the analysis of existing natural language-based documents, if such documents exist. The knowledge engineer, or whoever is responsible for building the knowledge base, has to interact with the domain expert to capture the essential knowledge so that these concepts can be well defined. To accomplish such a task, one must first try to fit a new concept which needs to be defined into the existing domain-independent ontology and application domain ontology. Once the concept is in the ontology, the property hierarchies of its immediate parent will function as templates to guide formulation of its own properties for this concept. Clearly such an approach makes the analysis process much easier.

It is often the case that the knowledge encoded into the knowledge base may be much more than that actually needed for the design purposes, since at the early stage we may not know which concepts and which properties will be needed during the design. However, such an effort is no means a waste of time. The extra knowledge one captures will enable the designer to have a better understanding of the problem domain, and also may be reused in the future.

The properties we may define for any particular concept may vary from very general to very specific. Normally the general properties a concept may have can be defined using the GenericProperties hierarchy. Of course the generic properties of the given concept are not limited to the items inherited from the domain-independent

ontology. Every concept can have their own properties defined beyond the inherited properties in any of its viewpoints. More specific properties confined to the specific application domain can be defined under the domain specific viewpoint. (In our example, this is the TelephoneDomainProperties (Figure 5.3)).

One of the most important aspects of domain knowledge that one may capture in a CODE knowledge base is that the knowledge can be independent of any design methodology or implementation language. This contrasts sharply with existing software engineering tools and technologies, which commit one to a design methodology or implementation language early on (e.g. OMT, C++ etc.). We see this independent approach as an advantage, since reuse will then be possible by other methodologies and implementation languages. The knowledge encoded should be easily understandable by any domain expert, who may or may not be an expert on any of the design methodologies or implementation languages, and even may not be a computer person at all.

To illustrate these ideas, we have defined in our CODE knowledge base an application concept hierarchy in the telephony domain, i.e. we have captured knowledge in about 70 concepts which are commonly used in the telephony domain, and that must be understood and agreed upon by all parties in a telephony design situation. It is, however, infeasible to discuss all these concepts here. We can only provide a simplified sub-tree in Figure 5.5 and Figure 5.6, and discuss some of the key concepts as examples.

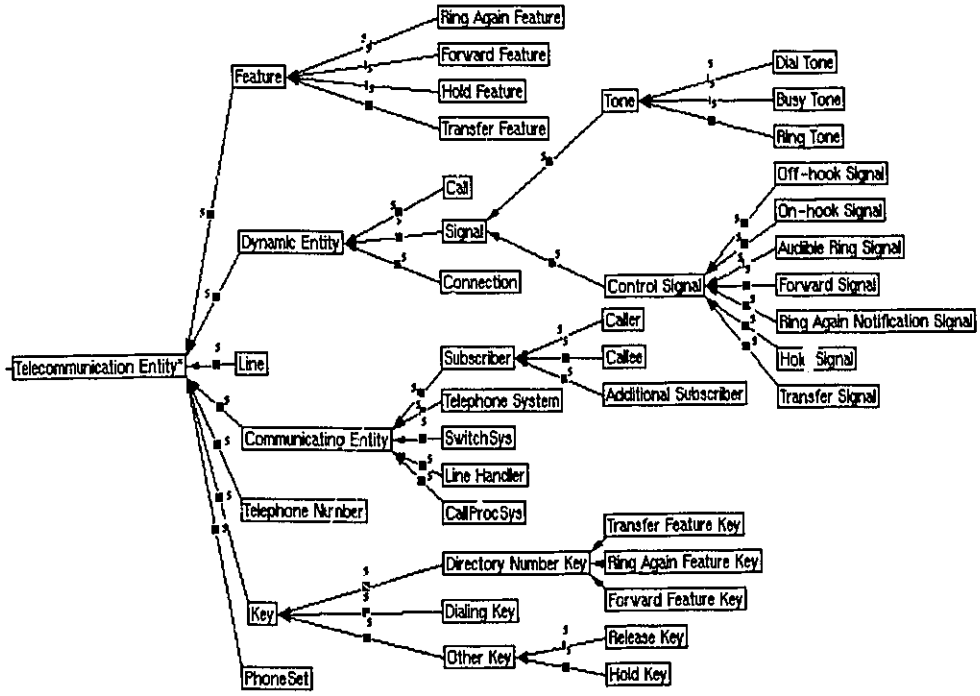


FIGURE 5.5 *Telephone System concept hierarchy defined in CODE knowledge base (the Telecommunication Entity branch)*

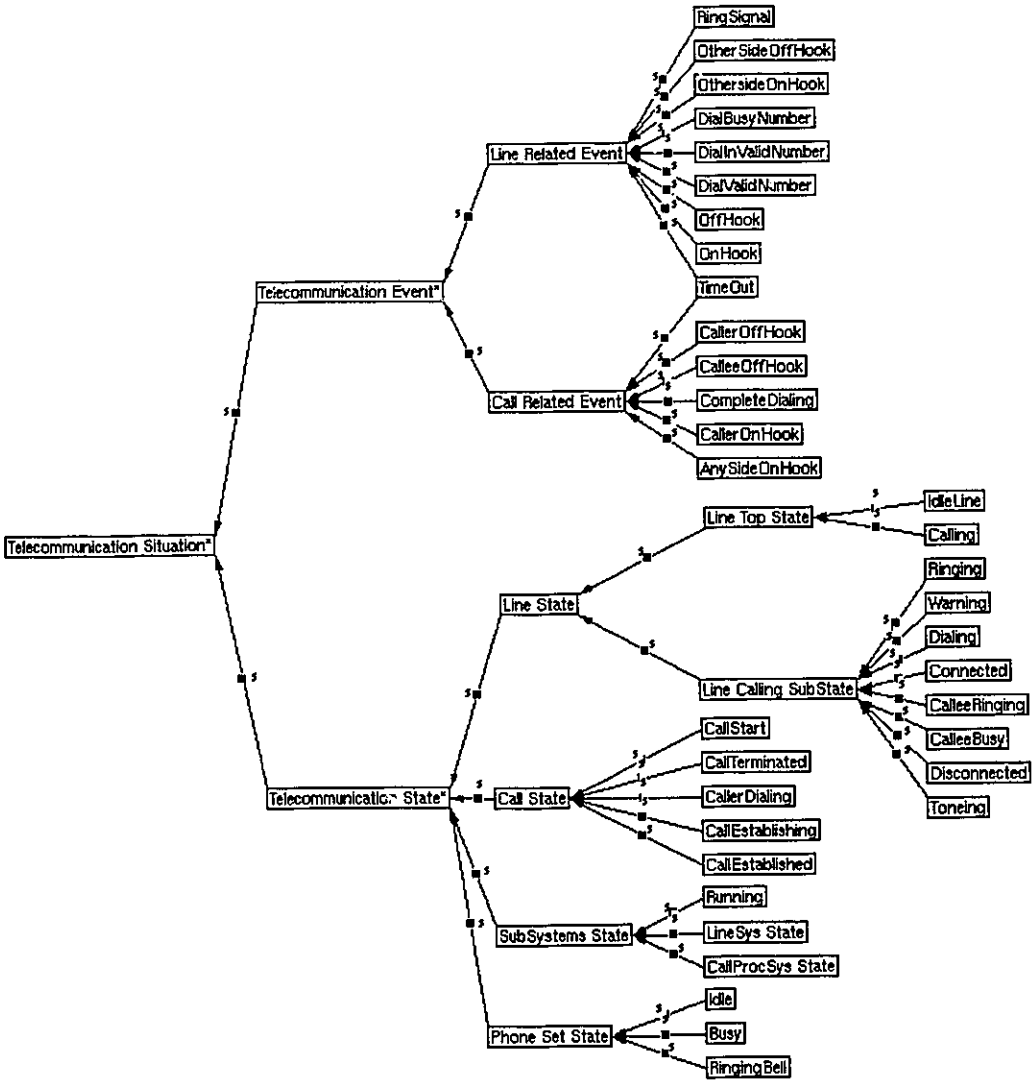


FIGURE 5.6 *Telephone System concept hierarchy in CODE knowledge base (the Telecommunication Situation branch)*

5.3.3.1 Telecommunication Entity

Telecommunication entities belong to the telecommunication domain, therefore, we use the multi-parenting feature of CODE to include all subconcepts of

telecommunication entity as children of the concept entity under the domain-independent ontology so that all these concepts can inherit the generic properties of entities (see Figure 5.2). Typical telecommunication entities are lines, phone sets, features, etc.

One kind of telecommunication entity are the *dynamic entities*, which are dynamically created or destroyed, i.e. they all have a start time and an ending time. Those properties are specific to the application, and defined under the application specific properties hierarchy (Figure 5.7). Examples of dynamic entities include signals, calls, and connections. They inherit all the common properties from the generic concept of entity.

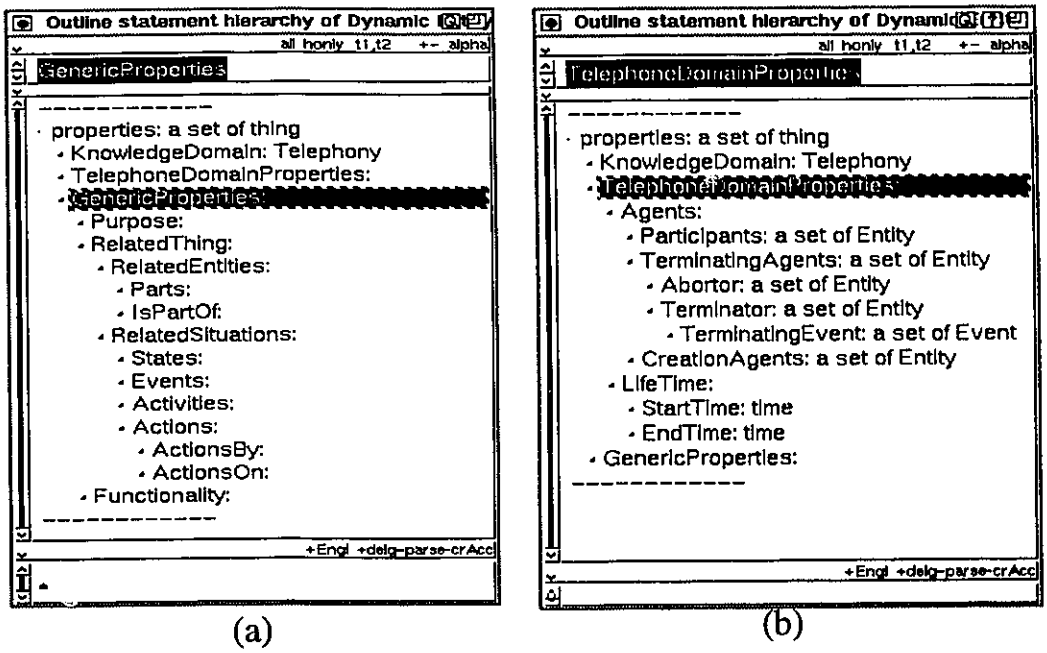


FIGURE 5.7 *Generic (a) and domain specific (b) properties of dynamic entities*

Another type of telecommunication entity are the *communicating entities*, which all have one or more entities they communicate with via some communication channel. A caller is a typical communicating entity: it communicates with one or more

callees through a switching system. Such telecommunication specific properties again are defined under the TelephoneDomainProperties hierarchy (Figure 5.8).

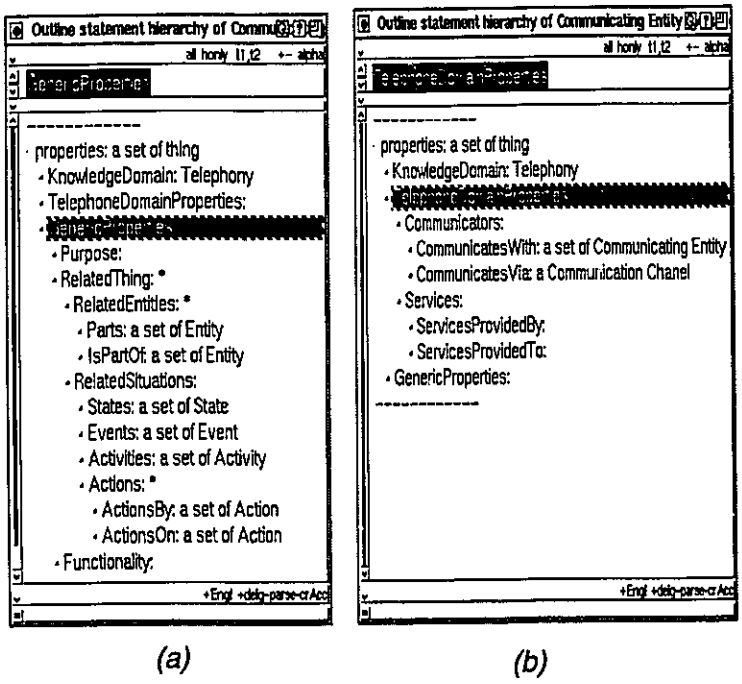


FIGURE 5.8 *Generic (a) and domain specific (b) properties of communicating entities*

We now describe a few more specific concepts, one in each of these subgroups (Dynamic Entity, Communicating Entity, and Telecommunication Entity) to illustrate how the properties are inherited and defined using our ontological framework.

Line (Telephone Line): As a subconcept of Telecommunication Entity, a line inherits all properties an entity may have as defined in the domain-independent ontology. Its purpose is to allow transmission of telecommunication signals. It has a related entity which is the telephone number, and it has states. These are common properties shared by any kind of entity. Each individual may have different values for some of these properties. Beyond those common properties

inherited from the generic concept of entity, a line has its own domain specific properties which is not shared by any other kind of entity. Those properties are defined under the TelephoneDomainProperties viewpoint. A line has an owner which is a subscriber, another telecommunication entity, and connects a telephone set and a switching system together. A line can be in service or not. Figure 5.9 is a summary of those properties defined in a CODE knowledge base.

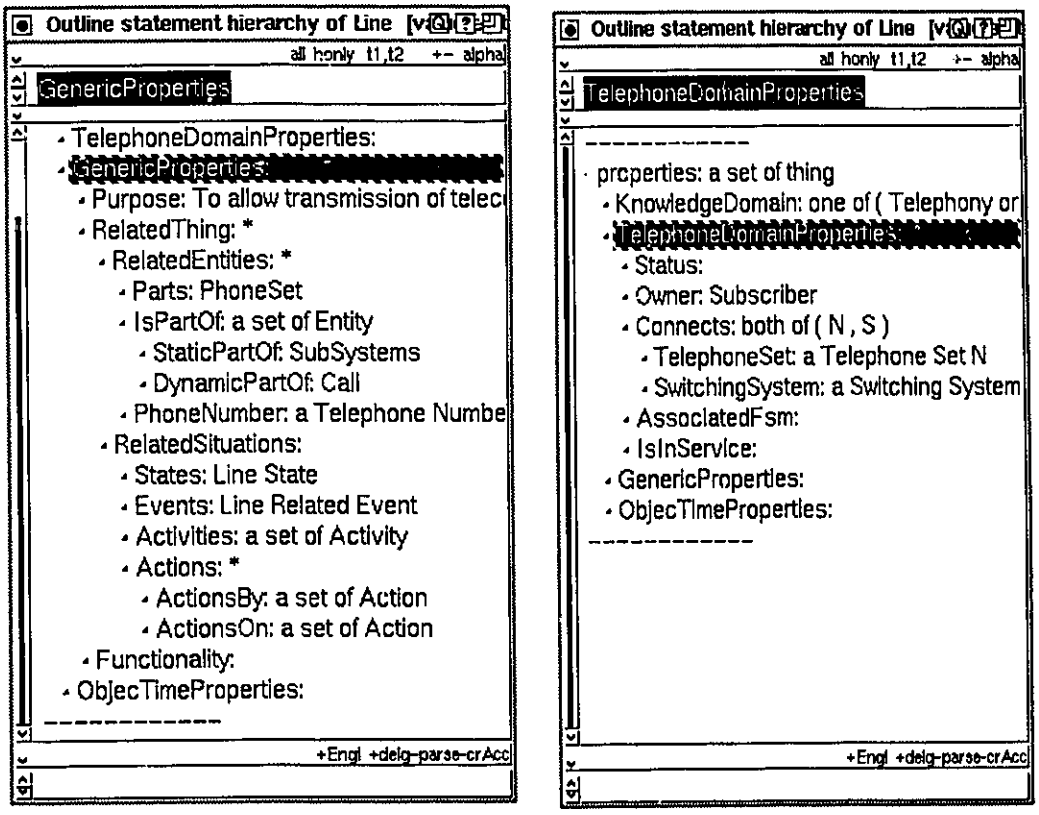


FIGURE 5.9 *Generic (a) and domain specific (b) properties (viewpoints) of a Line*

Call: A call is a dynamic entity which has a limited life time, i.e. a start time and an end time. It is dynamically created when the call starts, and destroyed when the

call terminates. Beyond the inherited properties from the concepts entity and dynamic entity, it has a number of specific properties (Figure 5.10). For example, it must have some participants which include an originating side, i.e. a caller, a destination side (callee), with a number of optional other participants. It can only be aborted by the caller, and can be terminated by either side.

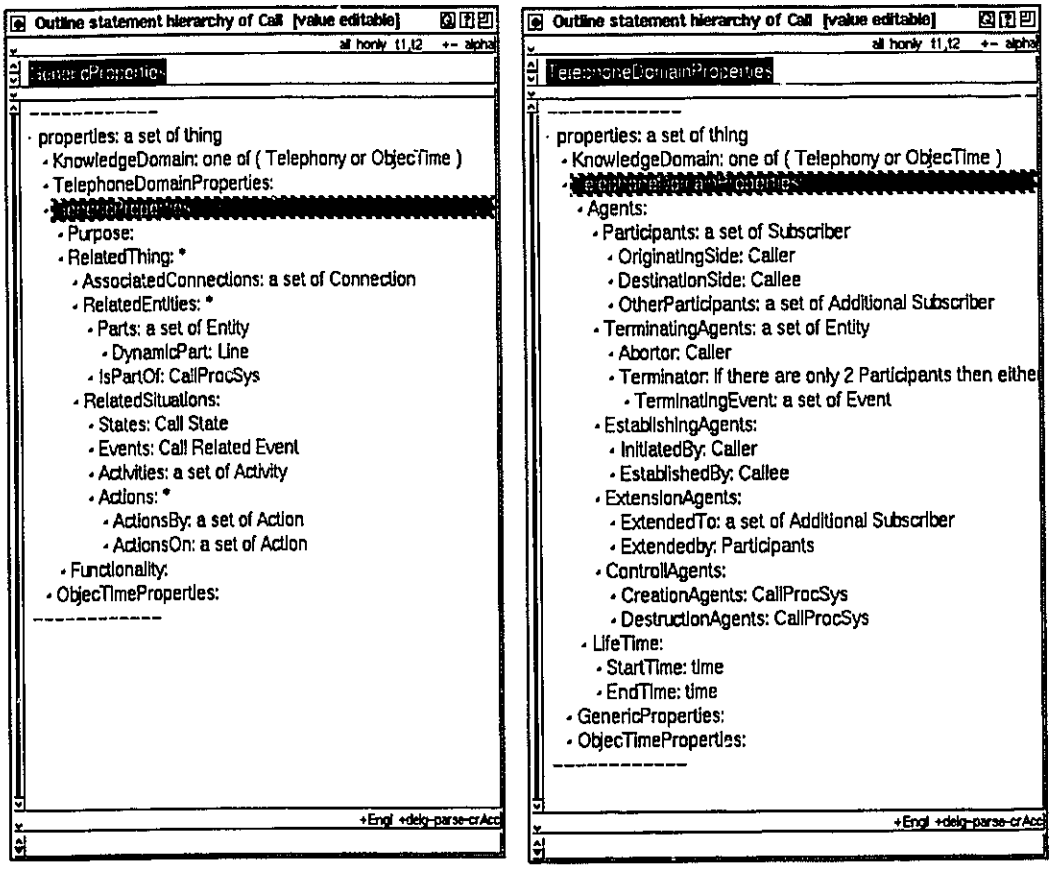
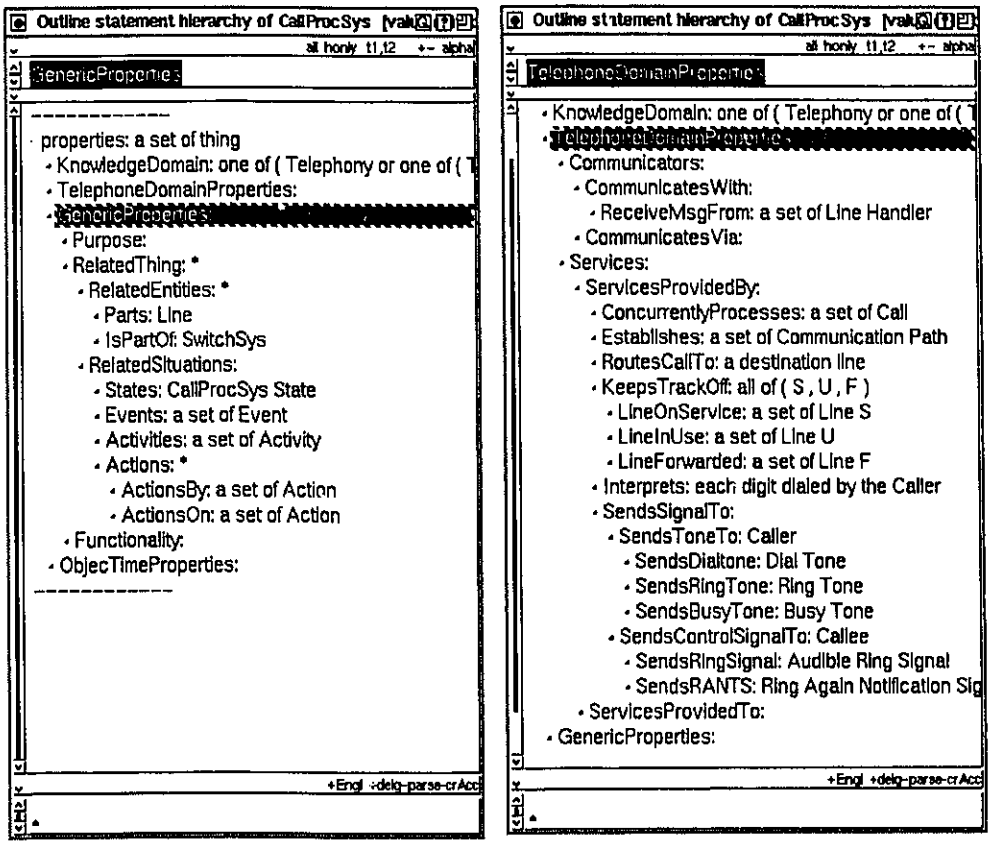


FIGURE 5.10 *Generic (a) and Domain specific (b) properties (viewpoints) of concept Call*

CallProcSys (Call Processing System): CallProcSys is a typical communicating entity which has a number of domain specific properties different from other telecommunication entities (Figure 5.11). It receives messages from a set of

line handlers, and provides a number of services, such as concurrently processing a set of calls, establishing a set of communication paths, routing calls to their destination lines, and keeps track of various line statuses. Detailed properties can be found in Figure 5.11.



(a) *(b)*

FIGURE 5.11 *Generic (a) and Domain specific (b) properties of concept CallProcSys (Call Processing System)*

5.3.3.2 Telecommunication Situation

While entities represent the structural aspect of the system under analysis, the concept of Situation captures the behavioral aspects of the system. We have derived an ontology about the generic concept of situation in Chapter 4, which becomes a part of

our ontological framework. Now we describe how a behavioral concept is related to a subconcept of entity, and how it is used to capture the behavior.

To capture the behavior of each component of the system, we have defined a number of behavioral properties under the GenericProperties hierarchy for every entity. The value of these properties may then serve as a pointer to another concept, which is normally a subconcept of situation. In our example, we employ a grouping concept Telecommunication Situation to organize these behavioral concepts. Each behavioral concept then must fit into the domain-independent ontology under the sub-tree of situation to inherit properties from this ontology. The same rules for defining entities are applicable for defining behavioral concepts. For instance, the value of the property States (Figure 5.10) for the concept Call is another concept, Call State. This property thus links the two concepts together. Similarly, the value of the property Events for the concept Call is Call Related Event. This property links the concept Call to the concept Call Related Event. As Call State is a subconcept of the concept State and Call Related Event is a subconcept of Event, these two concepts describe the dynamic behavior of the concept Call. A complete list of all the states that the concept Call may have, and all the events which may be involved in a Call, can be found in Figure 5.6.

Since Call is a dynamic entity, it has a starting state (CallStart) and a terminating state (CallTerminated). As we applied the rules described in a previous section, these two states are static in nature, therefore are sub-concepts of Static State in the domain-independent ontology. Other states are dynamic in nature, and therefore are subconcepts of Dynamic State.

Using CODE's graphical interface, we can illustrate the relationship among all the states and events which we have defined for concept Call, i.e. a state transition diagram (Figure 5.12). A similar approach can be applied to all other subconcepts of entities.

The mechanism described above can also support hierarchical state diagrams. Figure 5.13 illustrates the concept hierarchy of the concept Line State, which has as experiencer a Line, which means the hierarchy represents the state of a Line. All concepts under the grouping concept Line Top State at the left subwindow of the diagram are the first level states, while all concepts under grouping concept Calling

SubState are substates of state Calling. The relationship between the concept Calling and any of its substates is defined in the properties EncompassingSituations and SubSituations. For example, the concept Ringing, which is a substate of Calling, has a property under EncompassingSituation as SuperStates with a value Calling (Figure 5.13). In other words, this property can be read as "concept Ringing, a kind of Line State, has a SuperState which is Calling". Similarly, the concept Calling has a property SubSituations with a value of Calling SubState. This property is inherited by all the subconcepts of Calling SubState including the concept Ringing.

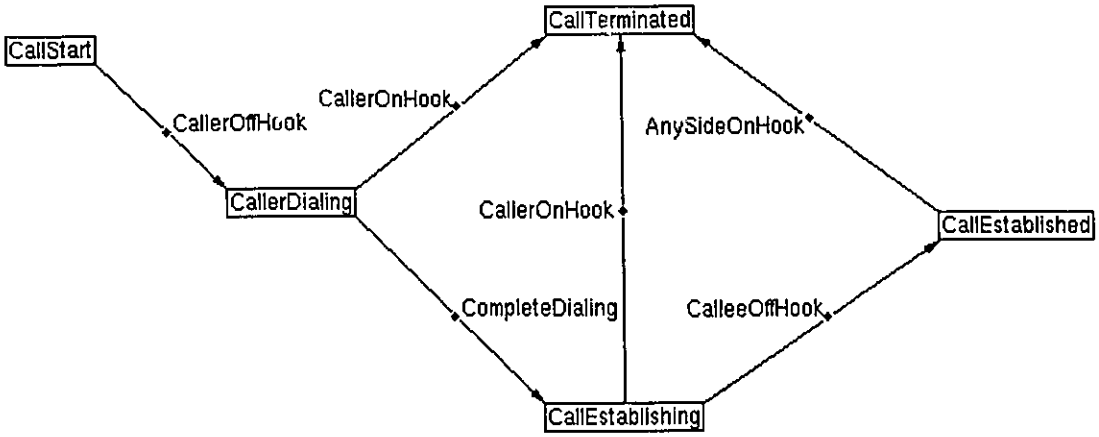


FIGURE 5.12 *State diagram of the concept Call*

Adopting the same approach for the concept Call, we can use CODE's graphical interface to illustrate the relationship among all the subconcepts of Calling SubState (Figure 5.14), which shows a state diagram of the refined state Calling. The highlighted state in the diagram, state IdleLine, in this example is the only state from the top level which is related to the state Calling. It may be a disadvantage to show a top level state on a substate diagram, but on the other hand it clearly indicates from which top level state the entity moves into which substate, and to which state the entity will move when it exits from the substate diagram.

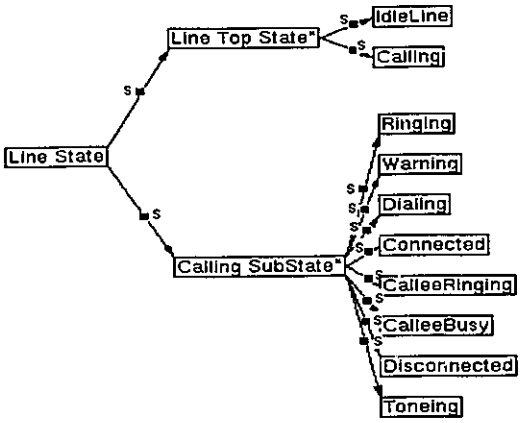


FIGURE 5.13 *Concept hierarchy of concept Line State*

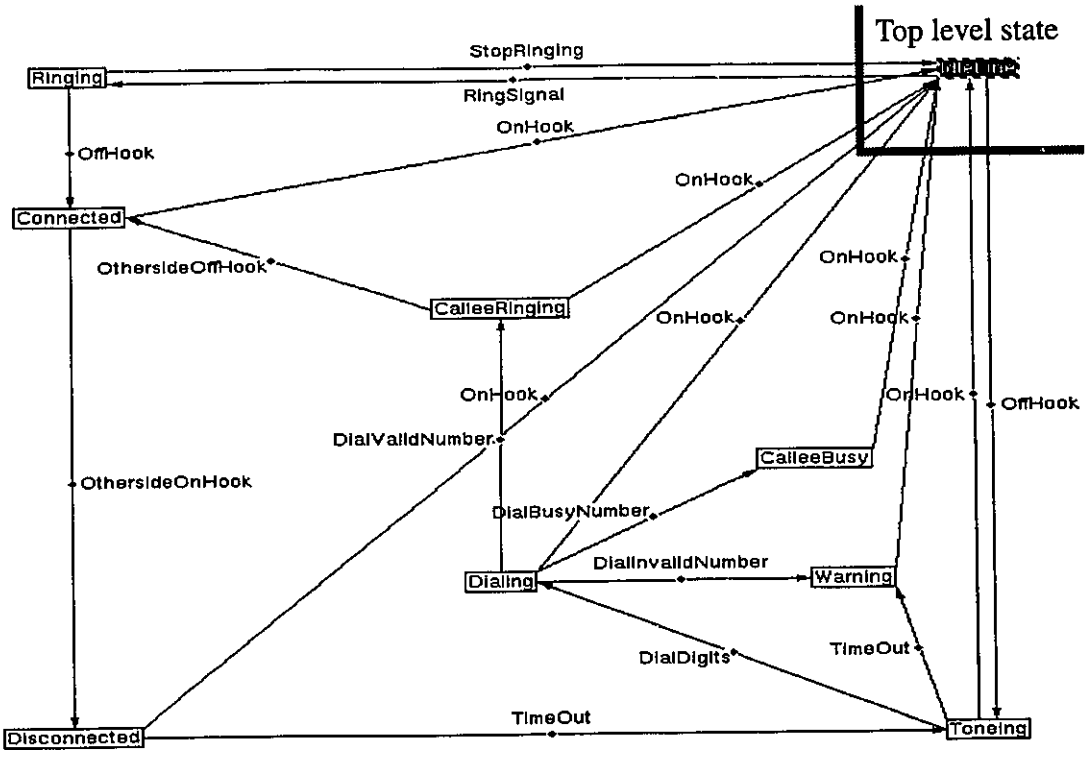


FIGURE 5.14 *Refinement of state Calling*

5.3.4 Design Knowledge

Software design is a complex process which involves a variety of knowledge-intensive activities, such as understanding of the domain knowledge, interpretation of the requirements and specifications, exploration and analysis of various design alternatives, consideration and reuse of existing components and solutions, identification of design goals and dependencies, and recording of design decisions. In this section we describe how CODE can be used as an intelligent assistant to manage the knowledge involved during the design process. We illustrate how to use CODE and ObjecTime to capture the essential design knowledge based upon the domain knowledge and requirements we have derived, and carry out the design.

5.3.4.1 Encoding Design Knowledge with CODE

Design decisions and knowledge can be independent of any implementation language or methodologies. Indeed, to maximize reuse and to be more flexible, they should be. However, they are dependent upon the design methodology which one chooses to perform the design. In our model, we have developed an ontological framework which allows knowledge of multiple design methodologies to be encoded into a single knowledge base, and one may carry out more than one design for one application. In our telephone system application example, we have selected ROOM/ ObjecTime as the design methodology to perform the design.

The first step in encoding the design knowledge is to determine those concepts which are essential for the design. As we have indicated in an earlier section, knowledge encoded under the application domain ontology is usually more than that necessary for the design; therefore, the designer has to decide which concepts are essential for the design. The guideline is the requirements. For example, in our telephone system knowledge base, we have captured a number of features a switching system may provide to its subscribers (Figure 5.5). However, as we plan to design a system which is not required to provide these advanced services at the moment, the concepts under Feature therefore are not essential for the current design task, though they may become design concepts in the future.

Upon the selection of a design concept, one must fit it into the methodology ontology so that properties of that methodology concept can be inherited by this

design concept and new values may be added to some of the properties. Those properties are grouped under the methodology viewpoint. In our example it called ObjecTimeProperties.

We employ the concept of Call again to illustrate how this process works. Figure 5.15 illustrates the position of the concept Call in the ontological framework. As a dynamic entity, Call can be considered as a perfect candidate for an Optional Actor, since it is dynamically created and destroyed by CallProcSys (see properties in Figure 5.10), i.e. it has a limited life time, and these are the essential properties of an optional actor (optional actor is a kind of dynamic actor, see *Optional Actor* on page 37). Indeed, the existing knowlege base may contain such design guidelines even as active rules: "Consider a Optional Actor for a Dynamic Entity". Thus someone unfamiliar with the design methodology can more easily learn it. Meanwhile, since we are designing a switching system which should be able to handle multiple calls simultaneously (see domain properties of CallProcSys in Figure 5.11 on page 91), this optional actor must be replicated at design time. Therefore, Call is also a perfect candidate for a Replicated Actor. In ObjecTime, each actor of a design must be in an actor class. Based upon this ontology, Call inherits its design properties from these three ObjecTime concepts (Figure 5.16).

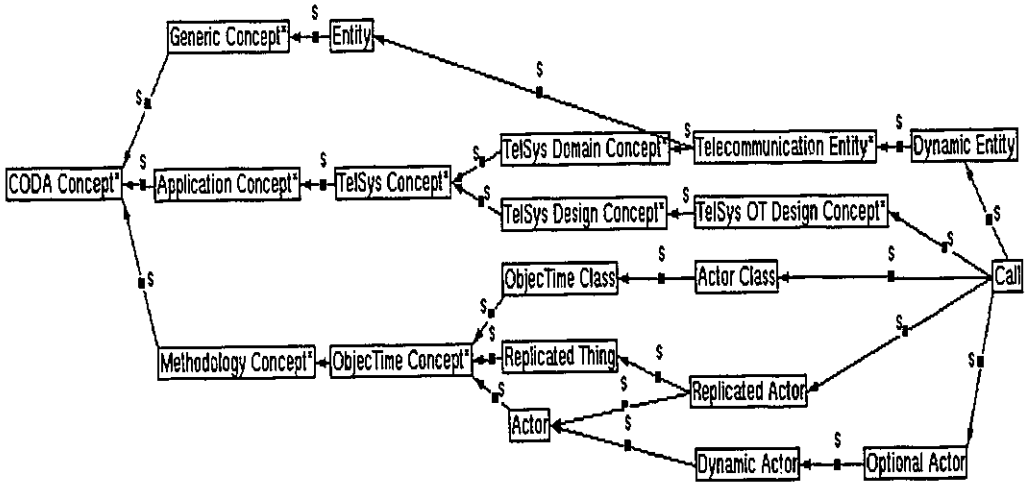


FIGURE 5.15 *Position of concept Call in the overall ontological framework*

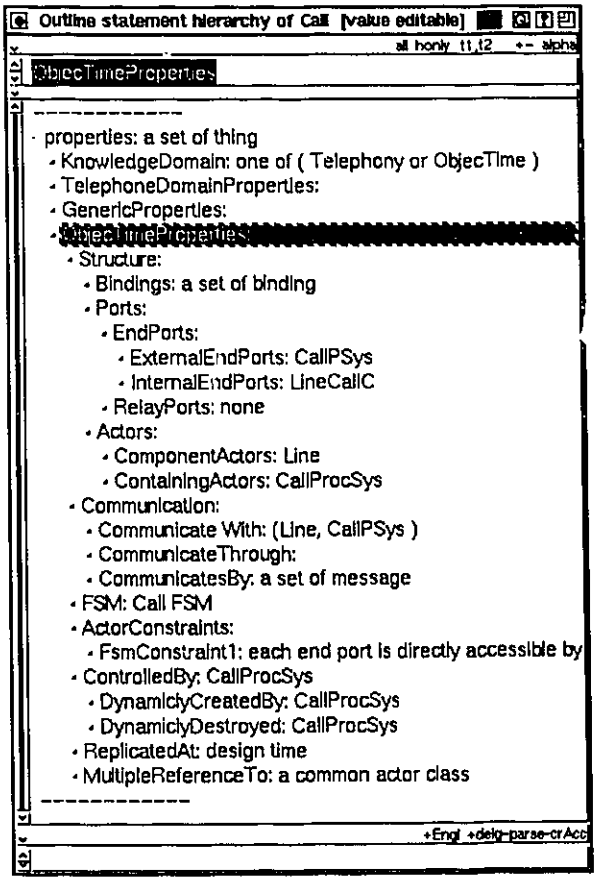


FIGURE 5.16 *Design viewpoint of concept Call using ROOM methodology*

Based upon the properties which design concept Call inherited from the methodology ontology, we then can complete the encoding of the design knowledge by assigning a proper value to each of these inherited properties according to the domain-independent and domain-specific properties, as well as defining its relation to other concepts. Figure 5.16 shows the properties Call inherits from the ObjecTime ontology. To complete the encoding of design knowledge, one only need to examine these properties one by one and decide what should be the proper value for each of these properties. For example, the values of the property Ports, which Call inherited

from the Methodology onotology, are *"a set of External End Port"* and *"a set of Internal End Port*[1]*"*. These indicate that as an actor under the ROOM/ObjecTime methodology, Call must have some ports as part of its structure. Therefore, we need to find out with who Call communicates with. From the generic and domain specific property hierarchies (Figure 5.10) we know that Call is part of CallProcSys and has a part which is a Line. Therefore, Call needs an External End Port to communicate with its containing actor CallProcSys, and an Internal End Port to communicate with its component actor Line. The meaning of this design decision is twofold: first, it implies that Line and CallProcSys should also be design concepts, and if they have not been defined so we should define them as design concept sooner or later; secondly, we need to define two new design concepts to represent these two ports, since the concept Port only exists within this specific design methodology. No concepts related to the concept Port have been defined during domain analysis. As a result, we define two new design concepts: CallPSys and LineCallC, and assign them to the property Ports of Call (Figure 5.16). Values of other design properties of Call are determined in the same way: values for the property Actors in Figure 5.16 are determined by the property RelatedEntities in generic property hierarchy (Figure 5.10a), and the values of the property ControlledBy in Figure 5.16 is determined by the property hierarchy of ControllAgents in the domain specific property hierarchy in Figure 5.10b. Encoding of design knowledge for other design concepts is similar to the process of encoding design knowledge for Call as we described.

During the encoding of design knowledge for potential actors, we can also identify the essential concepts of ObjecTime protocol classes and data classes. For example, as we assign a value of CallPSys to the ExternalEndPorts property of Call, we also identify CallPSysP as a protocol class, since every port must have a protocol defined.

To ease the design task, encoding of the design knowledge into a CODE knowledge base can be performed simultaneously with the actual design in ObjecTime, i.e. we do not have to follow the waterfall model to finish the encoding of

1. CODE cannot actively process plurals at the moment

design knowledge in CODE knowledge base first then to perform the actual design in ObjecTime, although we could if it is so desired. A better approach is to encode some of the design knowledge in to the knowledge base and then perform a partial design in ObjecTime. Then one may encode more new knowledge into the knowledge base and expand the design in ObjecTime, and so on. This process can be repeated any number of times. In other words, an iterative approach may be more desirable, since the encoded knowledge in the CODE knowledge base provides assistance to the design process in ObjecTime, while completing the design in ObjecTime provide feedback and inspiration to the encoding of design knowledge in CODE, particularly by actual execution.

5.3.4.2 Design in ObjecTime

Design in ObjecTime (or any methodology) is based upon the integrated understanding of the application domain knowledge and requirements with the ROOM / ObjecTime methodology knowledge. Each ObjecTime design starts with the identification of potential actors, communication protocols, data objects, event sequences, etc. The domain and design knowledge we have encoded in the knowledge base can now serve the purpose of guiding the design in ObjecTime. Comparing the design concept hierarchy we have defined in CODE (Figure 5.17 a) and the actual design in ObjecTime (Figure 5.17 b), we can see that the design in ObjecTime is a direct mapping of the design concepts in a CODE knowledge base. In this case, our design was strongly influenced by ObjecTime. Alternatively, we could have made a more abstract design, in which case the mapping would not be as straightforward.

The similarity between the design knowledge captured in our CODE knowledge base and the actual design in ObjecTime (Figure 5.17) reflects the design knowledge management process we proposed in the previous section. As the design knowledge is transferred into an ObjecTime design, the structural view of the design should be similar, otherwise the process of knowledge transformation is more difficult. This similarity also permits the design knowledge encoded in the CODE knowledge base to be treated as a generalized knowledge resource, since it can contain answers to queries such as what was the design rationale behind a particular design, what is the

relationship between one particular piece of the design and the rest of the system, etc. Of course this information may also be found out from the actual design in ObjecTime. ObjecTime indeed has a property browser to record any related knowledge in a natural language comment format for each actor and its state diagram. However, as we have indicated earlier in this thesis, there are many untreated knowledge management problems related to the natural language-based documentation. ObjecTime provides only this "commenting" to supply extra knowledge, while CODE treats both formal and commentary knowledge on an equal basis.

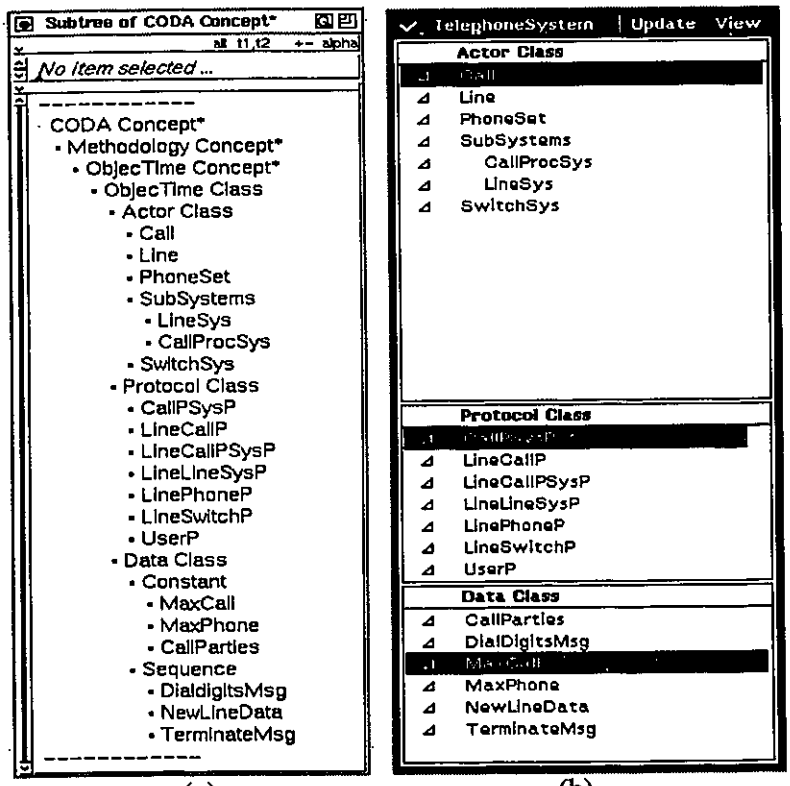


FIGURE 5.17 *Design concepts in CODE knowledge base (a) and the actual design in ObjecTime (b) (b is an actual ObjecTime window)*

Actor Structure

The structural or static aspect of an actor class which corresponds to a design concept in a CODE knowledge base can be found out from the ObjecTimeProperties viewpoint of the design concept. These properties not only define the structure of the corresponding actor, but also the relationship between this actor and other system components. For instance, the design properties of the concept Call has a structure which consists of an external end port known as CallPsys and an internal end port known as LineCallC. It has a containment relationship with Line and CallProcSys. These properties define the structure of actor Call in ObjecTime (Figure 5.18); they essentially what is shown by the diagram. In other words, the structural design of a Call actor can be described by the design properties of Call as a design concept when we encode the design knowledge into the CODE knowledge base. Comparing the structural design of a Call actor in ObjecTime (Figure 5.18) with the design properties called Structure in Figure 5.16, it is obvious that the static aspects of a Call actor can be fully described by the knowledge captured during the design knowledge encoding process in CODE. This is only one of many good examples showing how CODE in fact contains the same knowledge as ObjecTime.

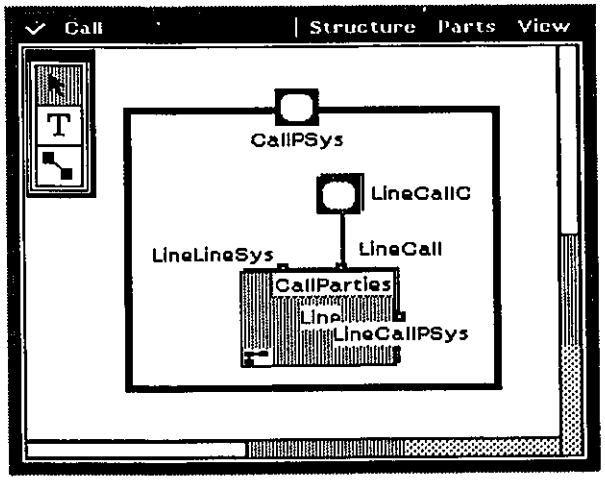


FIGURE 5.18 *Structure of actor Call as defined in an ObjecTime design*

Actor Behaviour

The behavior of an actor is defined by a finite state machine in ObjecTime. As we have described earlier, this has already been defined during the domain analysis phase for each entity which corresponds to an actor in ObjecTime when we capture the domain knowledge with CODE. Therefore, to define the dynamic behavior of a Call actor in ObjecTime, we can easily transfer the relationship among all Call-related states and events into an ObjecTime design. This could even be done automatically. Comparing the actual implementation of the FSM for actor Call in our ObjecTime design (Figure 5.19) with the relationship diagram (Figure 5.12) derived during the domain analysis phase, it is again obvious that the dynamic behavior of a Call actor can be fully described by the domain concept Call State in the knowledge base. Thus the task of actual design in ObjecTime becomes much easier after the conceptual analysis of domain concepts and "knowledge-level" design in CODE. Indeed, all knowledge, even the actual executable code, can be completely specified in CODE, and could be automatically "compiled" into ObjecTime.

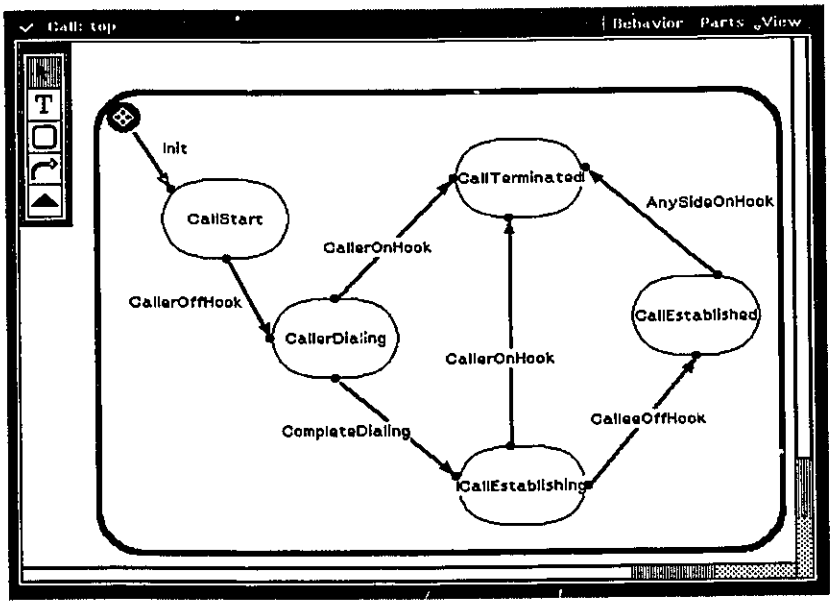


FIGURE 5.19 *State diagram of actor Call as defined in an ObjecTime design*

As the telecommunication system is a typical event-driven system, perhaps one of the most important and difficult tasks in an ObjecTime design is to "program" the transitions in a state machine, i.e. to specify the reaction of the actor upon receiving a message from one of its end ports, so that the design can be executed using ObjecTime's simulation feature. Figure 5.20 is an example of program for one of the transitions on a state diagram of Call actor. It says that when Call actor is in a CallerDialing state and the caller aborts the call by generating a CallerOnHook event, the Call actor sends a TerminateCall signal through its External End Port CallPSys to actor CallProcSys.

Executable knowledge, i.e. that is associated with transitions, can largely be defined in CODE by a group of properties and values which can automatically be translated into stubs for transition code. Figure 5.20 shows an example of such a partial specification of transition code, shown in Figure 5.21.

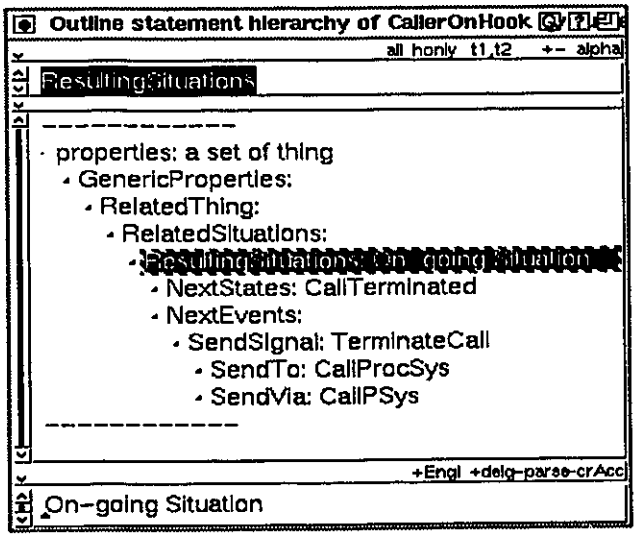


FIGURE 5.20 *Properties of concept CallerOnHook showing to which port the signal CallTerminated is sent*

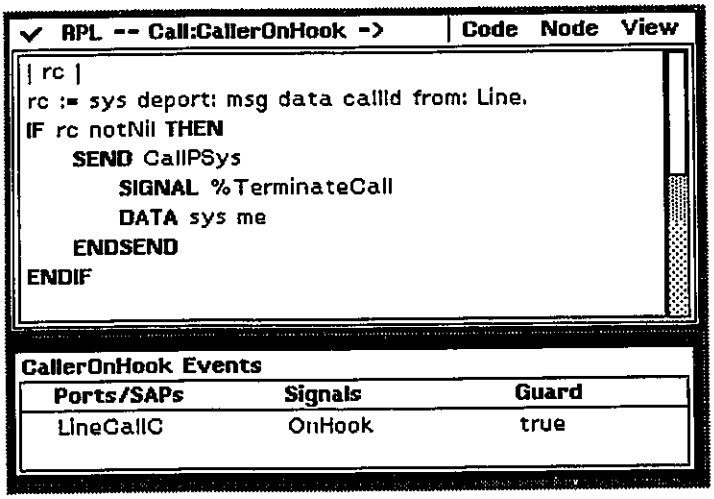


FIGURE 5.21 *Specification of event sequence on a state diagram transition for actor Call*

5.4 Discussion

We have illustrated with our simple telephone system example how management of domain and design knowledge is an important issue in software requirements analysis and design, and how it can be greatly assisted by a KMS such as CODE. The relationship between the knowledge management environment we described in this thesis and a software development environment was illustrated in Figure 1.1. The knowledge involved in a typical software development life cycle is captured in a knowledge management system. This knowledge then is used in a software development environment, in order to obtain the final products. These

products not only include the executable system implementation and associated supporting documents, but also the analysis and design knowledge of the system, i.e. the knowledge base, which may be reused in future software development.

During this experiment, we have experienced a number of difficulties related to the acquisition of domain and design knowledge. In this section, we provide a brief discussion of these issues.

It is reasonable to assume that one must have a good understanding of the application domain before s/he can actually design a system for the application. However, in many cases the software designer is not necessarily an application domain expert. Thus it is critical to provide the designer with enough domain knowledge so that the design task can be performed. Consulting natural language-based documentation is usually the main source of the information; consulting a domain expert is another source. Unfortunately there are many problems associated with these two sources.

The first problem is the lack of useful documentation for a particular application. In the first phase of our experiment, we were surprised to discover that there was not much documentation suitable for our purposes available about the general domain of telephony. We had to acquire necessary domain knowledge piece by piece from various sources and consultation with a domain expert.

The second problem is that existing documentation is often full of ambiguity, inconsistency, and incompleteness. To overcome these difficulties, it would be useful to have a document "analyzer" which would help the system analysts debug these problems and clarify the necessary domain knowledge[1]. Our experiment has illustrated that a KMS like CODE can assist in this task very well. We started to acquire the domain knowledge from various documents by encoding the acquired knowledge into a CODE knowledge base. The knowledge representation mechanisms

1. Dr. Skuce and Dr. Meyer are now working on such tools.

in CODE enabled us to discover and eliminate the inconsistencies and ambiguities encountered in those documents and derive a unified domain ontology.

The third problem with natural language-based documentation is that knowledge related with a particular concept is often buried all over the documents. It is very difficult to retrieve a "package" of related knowledge. In contrast, the CODE knowledge base we derived in our experiment provides a clear view of a particular concept by making such "packages" explicit. All knowledge related to a concept is captured in its properties and is displayed through the CODE property browser in the same place. The relationship between this concept and others are also clearly defined and easy to retrieve or display.

Another problem in domain knowledge acquisition is how to obtain agreement between domain experts and systems analysts as a team. Traditionally, the agreement is often derived by extensive brainstorming sessions with help of scratching the ideas or viewpoints on blackboard or paper. These agreements then may be documented in some kind of documents or minutes. In many cases, however, these agreements may never be documented in any form, but just kept in someone's mind. Knowledge derived by this kind of process however is volatile and easily lost or misunderstood. Instead of using blackboards or paper, we performed our brainstorming sessions with the help of our CODE knowledge base. The system provided us quick access to the existing knowledge entries and kept track of new agreements or ideas as new knowledge was discovered. It also functioned as a knowledge debugger during these sessions which enabled us to discover mistakes we had made at an early stage.

Beyond the difficulties during the acquisition of domain knowledge as described above, many difficulties may be encountered during the ObjecTime design phase if domain knowledge has not being captured or ObjecTime concepts are not well understood. The following paragraphs describe some of these problems.

One of the difficulties a designer may face is how to select and define the ObjecTime classes, especially when the designer is not familiar with the application domain. To carry out an ObjecTime design, the designer has to carry out an application domain analysis first. We have argued that CODE is a powerful tool which can provide

intelligent assistance to the analysts during this process. The design knowledge captured in the CODE knowledge base as we illustrated in our example can help a designer to decide which concepts should be implemented as an actor in the ObjecTime design, which protocol classes should be defined for the ports, which signals should be defined for which protocol classes, etc. Without the help of the CODE knowledge base, the designer still has to discover this knowledge and has to manage it by some other methods, usually "manually", which is error-prone since there is no dynamic linkage between the pieces of information.

To carry out an ObjecTime design of a reasonable complexity, one of the difficulties a designer may face is to have a clear picture of the event trace, i.e. we must know exactly which actor sends which signal to which other actor under what condition, and how that other actor will response upon receiving this signal. For example, does the actor which sends the signal have to wait until receiving response from the signal receiver, i.e. it is a synchronous messaging, or it does not wait at all, i.e. it is an asynchronous messaging? To answer all these questions, the designer has to figure out this event sequence and define it somewhere. There are two issues about this. The first is how the designer is going to figure out this event sequence. One way to do it is to adopt some methodology such as the event trace diagram approach suggested in [RUMB91], or by trying to figure out "use cases" as in the approach suggested in [JACO92]. The second issue is how and where to define this event sequence. A typical method is to define this sequence graphically or in text either on paper or somehow on-line. ObjecTime itself does not provide intelligent help to deal with event sequencing. To better deal with these issues, we propose a solution by using CODE to specify these signaling sequences, as we illustrated in Figure 5.21. This approach has several advantages. First, it is stored in a CODE knowledge base, therefore, it can be processed. Second, it is dynamically linked to the responsible actors since it is defined as properties of event which is dynamically linked to a design concept. Third, all the related signal sending sequences are dynamically linked with each other and can be visualized by a CODE concept relation diagram, similar to the approach by which derived our state diagrams. A special new diagramming mode could be added for event tracing.

In a typical ObjecTime design, an application contains multiple actors (Figure 5.17). However, the relationship of these actors is often not obvious. In other words, the architecture of the system is buried inside the structure of each actor, i.e. the overall architecture of the system is not visible. One has to image the overall structure of the system according to the structure of each actor. In our example, this difficulty is overcome easily by displaying a CODE relation diagram for any one of the actor classes under the property of Relationship (Figure 5.22).

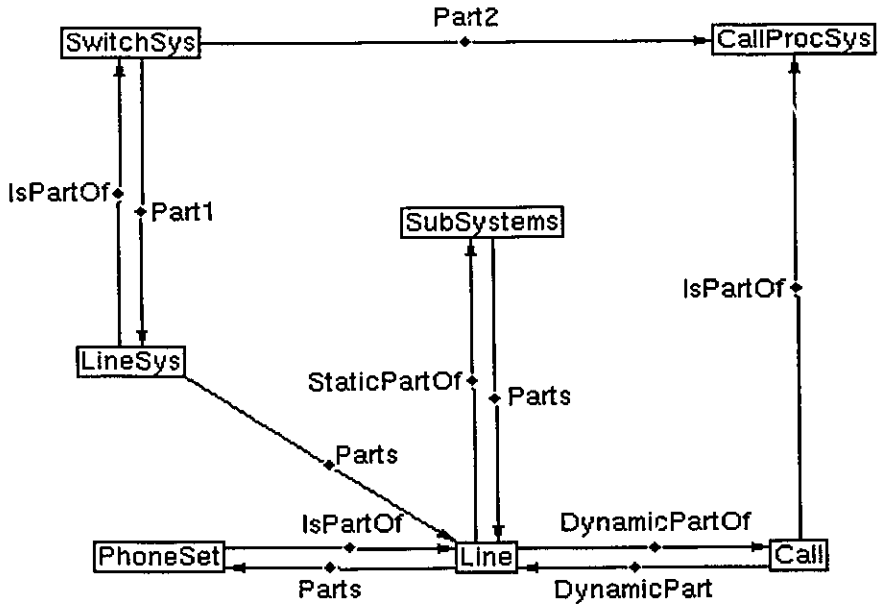


FIGURE 5.22 *Relationships between all actor classes in the telephone system design*

In terms of software reuse, we believe a ObjecTime design component can be better reused if these components are documented in a CODE knowledge base, since the latter provides a better view of the organization of these components, especially if the CODE and ObjecTime are dynamically linked together.

Another advantage of linking the two systems together is that ObjecTime can execute its design, which provide prototyping of the designed system. This execution not only validates the design in ObjecTime, but also validates the design knowledge encoded in the CODE knowledge base. Of course, physically linking two systems is not necessary, only desirable.

The CODE knowledge base constructed during the domain analysis and application design phase also serves as an on-line analysis and design documentation. Whenever it is desirable to retrieve the knowledge of a domain or design concept, the only work one needs to do is to open a property browser on the desired concept, which provides an structured representation of the required knowledge. With a small addition, CODE could generate simple English sentences automatically[1].

5.5 Summary

In this chapter we have proposed an ontological framework which allows integration of multiple software development methodologies and multiple applications into a single CODE knowledge base. The significance of this framework is actually not confined to CODE. One may reuse this framework in other environments too. It consists of three sub-ontologies: the domain-independent ontology, the methodology ontology, and the application ontology. We also defined some general procedures to support the knowlege encoding processes under this ontological framework.

A telecommunication example was developed to illustrate how a KMS like CODE can provide intelligent assistance to help software designers to carry out design tasks using sophisticated and advanced CASE tools such as ObjecTime. By looking at both the advantages and problems with our approach, we conclude that the application domain knowledge encoding process is a necessary step before one can apply

1. An earlier version, CODE2, did this.

ObjecTime to carry out an actual design, and the encoding of design knowledge with CODE can ease the difficulties of design in ObjecTime. The CODE knowledge base can also serve as an dynamic knowlege resource about a particular application and a repository for design reuse.

CHAPTER 6

Contribution and Concluding Observations

This chapter summarizes the major contributions and concluding observations we have drawn from this research. The last section provides a short list of possible areas for future work.

6.1 General Summary

To address a major problem, knowledge management, in software development, we began with the identification of some problems of knowledge management in the software development life cycle. We next compared knowledge engineering with software engineering and discussed how knowledge engineering might provide assistance to software engineering. Based upon our belief that knowledge management can become a major bottleneck in software engineering, we derived a classification of knowledge involved in software development into domain-independent knowledge, methodology knowledge, and application knowledge. As a result, we concluded that knowledge engineering can provide software engineers with a great deal of guidance and assistance in managing these three types of knowledge.

As part of a background study, a number of existing knowledge-based software development systems were reviewed in Chapter 2. Since we employed CODE and ObjecTime to carry out our knowledge management in the requirements analysis and design tasks, we provided a detailed review of both these systems. These two systems have certain useful features which were essential for our objectives in this study. We summarized these features and also the weaknesses of the systems in Chapter 3, which justified our selection of tools.

The finite state machine formalism is widely used in the software development practice and it is one of the best methods to describe the dynamic behavior of a system. However, there are no commonly agreed definitions of the key concepts in this formalism in the literature. Therefore, a conceptual analysis of the formalism was carried out in this thesis to clarify some of the key concepts in the formalism, especially the notion of state. This exemplified our approach: general ontological issues (concepts and terminology) must be clarified explicitly in a knowledge base before more specific knowledge can be organized.

The main objective of this thesis was to develop conceptually-oriented techniques for performing both requirements analysis and design, thereby deriving on-line high quality documentation in the form of a knowledge base. To illustrate how a KMS can assist software developers in these areas, we developed a telecommunication example to demonstrate these knowledge management techniques. CODE was used to carry out the knowledge management tasks, and ObjecTime was employed to carry out the final executable design.

6.2 Major Contributions

The major contributions of the thesis may be summarized as follows:

- Identification of some major knowledge-related problems in software development and how knowledge engineering can alleviate them.
- An ontology of the behavior-related concepts, including clarifying the notion of state and activity.
- A general ontological framework for knowledge management in the software development life cycle. The framework includes three ontologies: the domain-independent or generic ontology, the methodology ontology, and the application domain ontology. This ontological framework allows integration of the knowledge from multiple methodologies and multiple applications into one single knowledge base. The domain-independent and methodology ontologies can be used as templates to assist capturing application domain and design knowledge.
- An example illustrating how the CODE system and knowledge base may be used to guide the final design, in this case using a sophisticated CASE tool such as ObjecTime.

6.3 Concluding Observations

The major conclusions of the thesis can be summarized as foliows:

- Development of software engineering technology must include solving its knowledge management problems since software engineering is a knowledge intensive activity. Merging of knowledge engineering and software engineering may be a critical part of a solution to the software development crisis we are facing now.
- Different phases of software development requires different types of knowledge, and at the current state of art this knowledge is not integrated. It is our belief that an integrated environment to manage all types knowledge is necessary in order to provide maximum machine support for software developers. Our approach in this thesis demonstrates that this can be done.
- To have an integrated knowledge management environment, it is essential to develop a commonly agreed ontological framework that future software development can based upon, i.e. general ontological issues (concepts and terminologies) must be clarified explicitly before more specific knowledge can be organized.
- Conceptually-oriented domain analysis, which is often ignored in practice, is a very important step in the software development life cycle, because it clarifies the fundamental concepts and terminologies of the domain which all other phases of software life cycle depend upon.
- Knowledge management systems like CODE are powerful and efficient tools to provide machine support for conceptually-oriented analysis. Application domain knowledge captured with a KMS can be used to guide the acquisition of design knowledge which in turn can be used to guide the actual design using a CASE tool such as ObjecTime.
- A knowledge base developed during this development process can be used as an on-line requirements, design, or even implementation documents.
- As almost all CASE tools today have little knowledge management capability, integration of a KMS such as CODE with a CASE tool such as

ObjecTime or OMTool will provide software developers a powerful and productive development environment. Alternatively, the next generation of development tools should incorporate strong knowledge management capabilities. In other words, more effort will have to focus on the development of intelligent software development tools.

6.4 Possible Areas for Future Research

We now suggest some of the possible areas for future research which we believe are important. These areas are applicable to most of the knowledge-based software development systems we reviewed in this thesis, but some are more focused upon CODE.

- As we have indicated in Chapter 2, it is unfortunate that none of the existing knowledge-based software development systems can provide concurrent access, which is vital in a team development environment. Hence one of the areas for possible future research and development is to develop a mechanism for concurrency control so that a knowledge base can be developed and used by more than one person at a time. One of the solutions to this may be to use an object-oriented database system as the repository.

- Another possible area of future study is the development of mechanisms for knowledge base distribution and replication over a network. This should enable multiple developers to remotely access the knowledge base from different sites at the same time. An distributed database management system may be used to provide such a capability.

- A better knowledge import / export system must be developed so that knowledge stored in one system can be easily transferred to other systems. Currently most knowledge-based systems tend to stand alone without any knowledge transferring capabilities. It would be desirable if KIF-like syntax and semantics ([GENE92]) could be followed.

- More natural language processing capabilities should be developed for natural language-base document analysis.

- For systems such as CODE which already have some natural language processing ability, it may also be a good extension to have a natural language style output of the knowledge base.

- Although many knowledge-based systems have graphical user interfaces, these interfaces tend to emphasize knowledge entering while knowledge retrieving and searching interfaces are usually weak. For example, CODE has an excellent interface for knowledge encoding and browsing, but it is not convenient to search for a particular concept specified by its properties, though it indeed has some searching ability.

- As we have illustrated in our telephone system example, the CODE knowledge base developed during the domain analysis and design process may provide great help for the final design in ObjecTime. One of the possible areas for future study is to link the two systems together so that design concepts defined in CODE knowledge base can be automatically transferred into ObjecTime as a design element, or an ObjecTime design could be "reverse engineered" into a CODE knowledge base. Other CASE tools such as OMTool or programming environment such as Smalltalk and ObjectCentre could also be considered for such a connection.

References

[AMBR91] Ambriola, V., Ciancarini, P., Corrandini, A., and DeFrancesco, N., 1991. Towards innovative software engineering environments. J. Systems Software, 14: 17-29.

[BELA91] Belady, L.A., 1991. From software engineering to knowledge engineering: the shape of the software industry in the 1990s. International Journal of Software Engineering and Knowledge Engineering. 1 (1): 1-8.

[BOOC91] Booch, G., 1991. Object oriented design with applications. The Benjamin / Cummins Publishing Company, Inc., 580 pp.

[BORG89] Borgida, A., Brachman, R.J., McGuinness, D.L., and Resnick, L.A., 1989. CLASSIC: a structural data model for objects. Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data.

[BRAM91] Brachman, R., McGuinness, D., Patel-Schneider, P. and Resnick, L. 1991. Living with CLASSIC: When and how to use a KL-ONE-like language. In J. Sowa (ed.): Principles of Semantic Networks, San Mateo, Morgan Kaufmann.

[BRAS85] Brachman, R. and Schmolze, J. 1985. An overview of the KL-ONE knowledge representation language. Cognitive Science 9 (2): 171-216.

[DAV90] Davis, A.M., 1990. Software requirements: analysis and specification. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 516 pp.

[DEBE91] DeBellis, M., Sasso, W.C., and Cabral, G. 1991. Directions for future KBSA research. IEEE 0-8186-2605-4/91: 84-89.

[DEVA90] Devanbu, P., Selfridge, P.G., Ballard, B.W., and Brachman, R.J., 1990. A knowledge-based software information system. Proceedings of Conference on KBSA'90, Syracuse, New York, 110-115.

[DEVA91] Devanbu, P., Brachman, R., Selfridge, P., and Ballard, B.W., 1991. LaSSIE: a knowledge-based software information system. Communications of the ACM, 34 (5): 35-49.

[GENE92] Genesereth, M.R. and Nilsson, N.J., 1992 (eds.). Knowledge-interchange format. Version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University.

[GHAL93] Ghali, N. 1993. Managing software development knowledge: a conceptually-oriented software engineering environment (COSEE). Master's thesis, Department of Computer Sciences, University of Ottawa, 133 pp.

[GHE91] Ghezzi, C., et. al., 1991. Fundamentals of software engineering. Prentice Hall, Inc., Englewood Cliffs, 573 pp.

[GILL62] Gill, A., 1962. Introduction to the theory of finite-state machines. McGraw-Hill Book Company, New York, 207 pp.

[GREE83] Green, C., Luckham, D., Balzer, R., Cheatham, T., and Rich, C. 1983. Report on a knowledge-based software assistant, RADC TR 83-195, Contract No. F30602-81-C-0206, Kestrel Institute, Palo Alto, CA.

[GRUB93] Gruber, T.R., 1993. A translation approach to portable ontology specifications. Knowledge Acquisition 5: 199-220.

[HARE87] Harel, D., 1987. Statecharts: a visual formalism for complex systems. Science of Computer Programming 8: 231-274.

[HARE88] Harel, D., Lackover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., and Shtul-Trauring, A., 1988. STATEMATE: A working environment for the development of complex reactive systems. Proceedings of 10th IEEE International Conference on Software Engineering, Singapore, April 1988: 396-406.

[HARR88] Harris, D. and Czuchry, A., 1988. The knowledge-based requirements assistant. IEEE Expert, 3 (4).

[JACO92] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G., 1992. Object-oriented software engineering, a use case driver approach. Addison-Wesley Publishing Company. New York, 524 pp.

[JOHN88] Johnson, W.L., 1988. Deriving specifications from requirements. 10th International Conference on Software Engineering, April 11-15, 1988, Sigapore, 428-438.

[JOHN90] Johnson, W.L., Harris, D.R., and Sanders, L., 1990. Requirements analysis using ARIES: themes and examples. Proceedings of the 5th Conf. on KBSA 90, Syracuse, New York, 121-131.

[JOHN91] Johnson, W.L., Feather, M.S., and Harris, D.R., 1991. The KBSA requirements / specification facet: ARIES. IEEE 0-8186-2650-4/91: 48-56.

[JOHN92] Johnson, W.L., Feather, M.S., and Harris, D.R., 1992. Representation and presentation of requirements knowledge. IEEE Transactions on Software Engineering, 18 (10): 853-869.

[LANE90a] Lenat, D. and Guha, R. 1990. Building large knowledge based systems. Reading, MA, Addison Wesley.

[LANE90b] Lenat, D., Guha, R.V., Pittman, K., Pratt, D., and Shepherd, M., 1990. CYC: toward programs with common sense. Communications of ACM, 33: 30-49.

[MOTE88] Motta, E., Eisenstadt, M., Pitman, K., and West, M. 1988. Support for knowledge acquisition in the Knowledge Engineer's Assistant (KEATS). Expert Systems 5 (1): 21-50.

[MOTR91] Motta, E., Rajan, T., Domingue, J. and Eisenstadt, M. 1991. Methodological foundation of KEATS, the Knowledge Engineer's Assistant. Knowledge Acquisition 3: 21-47.

[OBJ92] ObjecTime Lt., 1992. ObjecTime User's Manual. ObjecTime Limited, Kanata.

[PORT88] Porter, B.W., Lester, J., Murray, K., Pittman, K., Souther, A., Acker, L., and Jones, T., 1988. AI research in the context of a multifunctional knowledge base: the Botany knowledge base project. AI88-88, Department of Computer Science, University of Texas at Austin.

[PUNC88] Puncello, P.P. et al. 1988. ASPIS: a knowledge-based CASE environment. IEEE Software, March 1988: 58-65.

[REUB89] Reubenstein, H. and Waters, R., 1989. The Requirements Apprentice: an initial scenario. Proc. 5th Int. Workshop Software Specification and Design. Washington, D.C.: IEEE Computer Society Press, 211-218.

[REUB91] Reubenstein, H. Waters, R., 1991. The Requirements Apprentice: automated assistance for requirements acquisition. IEEE Transactions on Software Engineering 17(3): 226-240.

[RICH89] Rich, C. and Waters, R., 1989. The Programmer's Apprentice: a research overview. IEEE Computer 21 (11): 10-25.

[RICH92] Rich, C. and Waters, R., 1992. Knowledge intensive software engineering tools. IEEE Transactions on Knowledge and Data Engineering 4 (5): 424-430.

[RUMB91] Rumbaugh, J., et. al., 1991. Object-oriented modeling and design. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 500 pp.

[SCHO88] Schoen, E., Smith, R.G., and Buchanan, B.G., 1988. Design of knowledge-base systems with a knowledge-based assistant. IEEE Transactions on Software Engineering, 14 (12): 1771-1790.

[SELF90] Selfridge, P., 1990. Integrating code knowledge with a software information system. Proceedings of Conf. on KBSA 90, Liverpool, New York, 183-195.

[SELI91a] Selic, B., Gullekson, G., McGee, J., and Engelberg, I., 1991. ROOM: an object-oriented methodology for developing real-time systems. CASE'92 Fifth International Workshop on Computer-Aided Software Engineering, Montreal, Quebec, Canada, July 6-10, 1992.

[SELI91b] Selic, B. et al., 1991. Object-oriented design concepts for real-time distributed systems. Real-time and Embedded Systems Workshop, ACM Conference on Object-Oriented Programming, Systems, and Languages (OOPSLA)'91, October 1991, Phoenix AZ.

[SELI92] Selic, B., Gullekson, G., McGee, J., and Engelberg, I. 1992. ROOM: an object-oriented methodology for developing real-time systems.

CASE'92 Fifth International Workshop on Computer-Aided Software Engineering, Montreal, Quebec, Canada, July 6-10, 1992.

[SELI93] Selic, B., 1993. An efficient object-oriented variation of the statecharts formalism for distributed real-time systems. CHDL '93: IFIP conference on Hardware Description Languages and their Applications, April 26-28, 1993, Ottawa, Canada.

[SELI94] Selic, B., Gullekson, G., and Ward, P., 1994. Real-time object-oriented modeling. John Wiley and Sons, 477 pp.

[SKUC91] Skuce, D. 1991. Knowledge management in software engineering design: a tool and an experiment. Department of Computer Sciences, University of Ottawa, 13 pp.

[SKUC92] Skuce, D., 1992. A knowledge representation for interactive knowledge management. AI Laboratory Report, Department of Computer Science, University of Ottawa, 51pp.

[SKUC93a] Skuce, D. 1993. CODE concepts and conventions. Artificial Intelligence Laboratory, University of Ottawa, 15 pp.

[SKUC93b] Skuce, D. 1993. CODE version 4.1, a system for knowledge management: acquisition, analysis, and retrieval. Artificial Intelligence Laboratory, Dept. of Computer Science, University of Ottawa, 5 pp.

[SKUC93c] Skuce, D. 1993. A multi-functional knowledge management system. Knowledge Acquisition, 5: 305-346.

[SKUL93] Skuce, D. and Lethbridge, T. 1993. A knowledge representation for interactive knowledge management. In preparation, 52 pp.

[SKUL94] Skuce, D. and Lethbridge, T. 1994. CODE4: a multifunctional knowledge management system. In: Proceedings of the 8th Knowledge Acquisition for Knowledge-based Systems Workshop, Banff.

[SKUM90] Skuce, D. and Monarch, I. 1990. Ontological issues in knowledge base design: some problems and suggestions. In: Proceedings of the 5th Knowledge Acquisition for Knowledge Based Systems Workshop, Banff.

[SKUM91] Skuce, D. and Meyer, I. 1991. Concept analysis and terminology: a knowledge-based approach to documentation. Proceedings of the 6th Knowledge Acquisition for Knowledge Based Systems Workshop, Banff.

[SOWA84] Sowa, J. 1984. Conceptual structures: information processing in mind and machine. Reading, MA, Addison Wesley.

[TSEA92] Tsai, J.J., Weigert, T., and Jang, H.-C., 1992. A hybrid knowledge representation as a basis of requirement specification and specification analysis. IEEE Transaction on Software Engineering, 18 (12): 1076-1100.

[WIRF90] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., 1990. Designing object-oriented software. Prentice-Hall, Inc., 341 pp.