Summer 1993

A Framework for Data Sharing in Computer Supported Cooperative Environments

Mohamed Youssef Eltoweissy
Old Dominion University

A FRAMEWORK FOR DATA SHARING IN COMPUTER SUPPORTED COOPERATIVE ENVIRONMENTS

by

Mohamed Youssef Eltoweissy
B.S. June 86, Alexandria University, Alexandria, Egypt
M.S. July 89, Alexandria University, Alexandria, Egypt

A Dissertation Submitted to the Faculty of Old Dominion University in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

Computer Science

Old Dominion University

August 1993

Approved by:

Dr. Hussein Abdel–Wahab, Advisor

Dr. Ravi Mukkamala

Dr. Stewart Shen

Dr. Shunichi Toida

Dr. Hany El–Sayed

To My Parents

Abstract

Concurreny control is an indispensable part of any information sharing system. Cooperative work introduces new requirements for concurrency control which cannot be met using existing applications and database management systems developed for non-cooperative environments. The emphasis of concurrency control in conventional database management systems is to keep users and their applications from inadvertently corrupting data rather than support a workgroup develop a product together. This "insular" approach is necessary because applications that access the database have been built with the assumptions that they have exclusive access to the data they manipulate and that users of these applications are generally oblivious of one another. These assumptions, however, are counter to the premise of cooperative work in which human-human interaction is emphasized among a group of users utilizing multiple applications to jointly accomplish a common goal. Consequently, applying conventional approaches to concurrency control are not only inappropriate for cooperative data sharing but can actually hinder group work. Computer support for cooperative work must therefore adopt a fresh approach to concurrency control which does promote group work as much as possible, but without sacrifice of all ability to guarantee system consistency. This research presents a new framework to support data sharing in computer supported cooperative environments; in particular, product development environments where computer support for cooperation among distributed and diverse product developers is essential to boost productivity. The framework is based on an extensible object-oriented data model, where data are represented as a collection of interrelated objects with ancillary attributes used to facilitate cooperation. The framework offers a flexible model of concurrency control, and provides support for various levels of cooperation among product developers and their applications. In addition, the framework enhances group activity by providing the functionality to

implement user mediated consistency and to track the progress of group work. In this dissertation, we present the architecture of the framework: we describe the components of the architecture, their operation, and how they interact together to support cooperative data sharing.

Acknowledgments

First, I thank God for His countless bounties and for directing me along the route to every success I reached and may ever reach.

My deepest appreciation goes to my advisor, Dr. Hussein Abdel-Wahab, for giving me the opportunity to become involved in research on computer supported cooperative work. It has been a fruitful experience. His insight, constructive comments, and valuable suggestions are reflected in the work described in this dissertation. I feel even more obliged to thank him for our excellent interaction and mutual understanding.

I would also like to express my sincere gratitude to the members of my committee Drs. Ravi Mukkamala, Stewart Shen, Shunichi Toida. Their genuine advice and discussions were essential for the development of this study. A word of appreciation also goes to Dr. Hany El-Sayed for his participation in the committee.

Special thanks are due to Dr. Larry Wilson, chairman of my examination committee, who generously offered me guidance and counseling while pursuing my research.

I also wish to extend my thanks to all my friends. In particular, I would like to thank Ashraf Wadaa and Osman Zeineldine for the ideas that emerged from our discussions, and Nahil Sobh, Ferasat Shah, Ibraheem Sharafeldine, and Hamid Oloso for the wonderful time we spent together in meetings, exercises, and travels.

The words fall far short from expressing my indebtedness to my beloved parents, my brother, and my sisters for their constant encouragement, endurance, and emotional support.

Finally, this work would not have been possible without the unfading support and perseverance of my wife and the overwhelming love and tender of my sons Youssef and Abdarrahman who made my everyday life more enjoyable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *Group work is a natural context for our activity: we benefit from prior work of other people, we cooperate actively with colleagues, we exchange views and participate in discussions, we engage in joint decision making, we communicate our completed work to others, and so forth.*

It is this context in which computer systems and their associated software are used. Yet, most existing software applications are developed to support only individual work in isolation. Little or no support is provided for communication, coordination, and information sharing activities that users are often engaged in. Hence, there is a legitimate need for computer facilities that understand and support these group activities.

Recent technological innovations in portable computing, user interfaces, and computer networking make it feasible to explore and develop new computer facilities that will help us work together more efficiently and conveniently [52]. The field that deals with the development of such facilities and its relevant research issues is generally termed **Computer Supported Cooperative Work** [26].

A principal concern in computer supported cooperative work (or CSCW) is how to store, maintain, and access data in work group settings. This dissertation argues that existing applications and the database management systems they use are inadequate for data sharing in cooperative environments; in particular, product development environments such as computer-aided design and manufacturing (or CAD/CAM) and computer-aided software engineering (or CASE). In a nutshell, cooperative product development will require each user to be as *"aware"* as possible of other users actions. This concept cannot be offered by existing applications and the database management systems they use. Supportive arguments are also included in [28, 5, 40, 31, 42, 44, 16, 6, 68, 17, 33, 56, 12], to name a few.

Our research investigates the data sharing requirements of cooperative product development environments. After identifying the restrictions imposed by conventional applications and database management systems, we propose a new framework that alleviates some of these restrictions and provides data sharing functionality needed to support cooperative development efforts.

The work in this dissertation presents the architecture of the framework; it describes the components of the architecture, their operations, and how they interact together to support data sharing in cooperative product development environments.

1.1 Motivation

The development of complex artifacts presents a strong case for the necessity of cooperation [48, 5, 6, 68]. Product development projects, such as VLSI design or software development, involve a group of developers working together to accom-

plish a common goal, which is the overall product that integrates the work done by different members of the product development group. Cooperation is necessary because no single developer has sufficient expertise, resources, or information to carry out a large scale project. Also, different developers may have different expertise for performing parts of the overall product.

Complex products are usually divided into simpler partitions, which can themselves be further divided resulting in a hierarchy of sub-products. Work on the sub-products is then distributed among members of the product development group. Each group member may be responsible for only part of the overall product. Group members cooperate, sharing the results of their activities as the overall product emerges from the results of the sub-products. Following this approach to product decomposition, *members of the product development group will work on parallel but related aspects of the product.*

A session within a product development project would consist of the steps taken by a product developer at a workstation using applications, such as a graphical editor or a circuit simulator, to manipulate (inter-related) objects in the database. The sessions are generally long and interactive, and their content may be dynamically determined and incompletely pre-specified. That is, the sequence of operations in a session is not a program that is defined statically or specified precisely before the product developer begins working. *The work in product development is creative, experimental, incremental, and iterative.*

During the course of the project, contributions will come from developers in different areas of specialization. These developers will interact with each other, and with the database, in order to exchange information pertinent to the substance

of their work (e.g., the common set of database objects, comments, and questions), the procedures of their work (e.g., the common view of the development process established by agreement on sharable paradigms), and the interpersonal relationships that underlie the work project (e.g., the possibility that one partner is not pulling his or her fair share). As a result of this information exchange, *different product developers will have some degree of awareness of each others' work.*

Product developers usually perform their tentative work in their local (or private) workspaces. They release their contributions to other members of the group at intervals rather than continuously. Furthermore, due to the size of the project, product developers cannot always be fully aware of the impact of changes they make on the global consistency of the product; aspects of consistency are defined by requirements, constraints, rules of design, policies, etc. As a result, the efforts of one product developer may conflict with those of another. Hence, *members of a product development group are typically concerned about the timely availability of information related to the project and about how the decisions made by others influence their current work.*

The conflicts that arise among product developers must be resolved in order to advance the current state of the product to the next refinement level. In general, a situation of conflict, in a product development environment, is one in which it seems temporarily impossible to have a consensus view among product developers as to what a part should look like. An important aspect of cooperative product development is that the willingness to cooperate facilitates the conflict resolution process. *Agreement is usually reached between product developers in discord through negotiation where intervention by the Project Leader may help in resolving the conflict.*

The question now is: Do conventional applications and the database management systems they use provide adequate support for cooperative product development? The answer is NO.

Evidently, the concept of *"work integration"* and the *"awareness"* property intrinsic to cooperative product development stand in sharp contrast with the assumptions that users are *"unrelated"* and *"isolated"* from one another, which underlie most conventional applications and database management systems; the conventional approach is at once too restrictive and inadequate for the needs of cooperative work, in particular the need for cooperative data sharing.

Conventional applications work in isolation of one another. Applications have traditionally been built with the assumption that an application which accesses database objects has exclusive access to those objects. Designers of conventional applications did not consider the fact that other applications might be needed to perform operations on related aspects of the same product. Consequently, if an application has some data objects in its read set, other applications should not be allowed to change those objects concurrently. Otherwise, the integrity of the application's results might be adversely affected. It follows that, at any given time, the applications that one user can employ strongly depend on which applications are presently in use by this and by other users. But this is quite restrictive in a cooperative product development environment, where product developers may have multiple applications run concurrently to complete the product as a team. A new approach is, therefore, needed in which an application would react to changes to its read set due to concurrent operations by other applications.

Likewise, conventional database management systems also go to great lengths to isolate people from one another in order to reduce interference or premature release of changes. In general, conventional database management systems use transactions as the unit of interaction between an application and the database. The conventional approach to ensuring database consistency in face of concurrent access is to ensure that each transaction on its own preserves consistency, and that each transaction is atomic (i.e., indivisible) with regards to permanence, recovery, and concurrency control [19, 61, 7, 2, 25]. That is, the result of a transaction that commits are stable over time, the result of a transaction that fails are re-instated completely or not at all, and the concurrency control scheme interleaves the operation sequences of transactions to generate schedules that are serializable (i.e., equivalent to a serial schedule in which transactions are executed one at a time). Since a partially executed transaction may violate consistency constraints, its results are never revealed to other transactions. On the other hand, the results of a committed transaction are permanent and globally visible to any other transaction. If an operation of a transaction conflicts with another operation of a concurrently executing transaction, one of the transactions involved in the conflict is either suspended or aborted. If the decision is to abort a transaction, then all of its effects must be removed from the system.

The aforementioned criteria, adopted by conventional database management systems to preserve consistency, are well suited to business applications such as banking and airline reservation in which users are isolated and unrelated, transactions are relatively short programs that are statically defined and independent of each other during development and execution, and atomicity of transactions is of paramount importance. Conventional database management systems do not support any other kind of consistency preserving criteria, for example, verification

protocols for designs. Moreover, the transaction processing schemes employed by these systems are not tailorable by programmers to more closely suit the needs of a particular application [28, 68, 32]. If long, incremental, and interactive product development activities are managed in the same way, they can impose severe limits on concurrency and hinder group work.

We can now contrast some fundamental characteristics of cooperative product development activities with those of conventional database transactions.

- Changes made during a transaction are not visible to other transactions until the transaction commits. Shielding a user from seeing the intermediate states of others' transactions is, however, in direct opposition to the goals of cooperative product development, where there is the urge to make each developer's actions visible to others; two developers might be modifying parts of the same object concurrently with the intend of integrating these parts; in this case, they might need to view each others' partial results to make sure they are not modifying the parts in a way that would make their integration difficult.

- Conventional database management systems suspend and abort transactions in service of concurrency control, and use rigid standardized methods of conflict resolution. The long-lived, and dynamically determined product development activities, however, cannot be suspended or aborted without inefficiency and loss of a significant amount of work. The product developer would definitely oppose deleting all of the work that might have lasted for hours. He or she might, however, cooperate with other developers to reverse the effects of some operations explicitly in order to regain consistency [22].

- Non-serializable schedules may be accepted in a product development envi-

ronment, since the primary concern is the correctness of the product rather than the sequence of steps that led to the product [5]; developers may exchange shared objects back and forth in a way that cannot be accomplished by a serial schedule.

- In conventional database management systems, consistency constraints are enforced uniformly on all transactions at all times. In contrast, product development activities may involve constructing hypothetical future states, the enforcement of constraints on these future states may often be deferred.

- In the course of a large-scale project, product developers often examine a great deal of material which provides general background to their work. If this material is treated as *"read"* from the point of view of serializability, too many conflicts arise to be acceptable [31].

To summarize, in product development environments the need for cooperation prevails. Current product development environments use conventional applications and database management systems. The *"insular"* approach to data sharing adopted by conventional applications and database management systems, however, constrains cooperation and thus impedes the progress of development. Overcoming these limitations poses formidable challenges to researchers and developers of systems that support cooperative work; what is needed is a new approach to generate a shared environment that unobtrusively offers up-to-date group context and appropriate levels of awareness among individuals and groups. Hence, our motivation.

1.2 Objectives

The broader goal of our research is to provide computer support for cooperation among people working together to achieve their common goals. This entails the support for communication, coordination, and information sharing among different groups and among members of the same group. In this dissertation, we focus on data sharing in product development environments, where cooperation among distributed and diverse product developers is essential for success, and where the characteristics and requirements of cooperation cannot be satisfied using conventional applications and database management systems, as shown in the previous section. We aim at promoting parallel cooperating activities as much as possible, but without sacrifice of all ability to guarantee system consistency. Specific objectives are stated as follows.

- To find appropriate types, representation, and granularity for data and meta data present in the cooperative development process.
- To define a suitable representation model to capture, maintain, and support the integration and common visibility of products (and/or sub-products) as developers from different perspectives engage in product development using a suite of applications.
- To develop concurrency control mechanisms that acknowledge the nature of cooperative product development as lengthy, interactive, dynamically determined, and incompletely pre-specified.
- To develop facilities that actively support and control data sharing among applications and higher level interactions among cooperative developers, rather than only prevent them.

1.3 Contributions

Toward our objectives, we further investigated the characteristics of cooperative product development environments, identified several new requirements for data sharing in these environments, and generated a list of desired features that would provide the specific requirements. We then aimed at developing enough conceptual structure and mechanisms to exhibit these features. The outcome of this research includes the following.

- **An extensible object-oriented data model suitable for cooperative product development environments**

  Objects in the model have descriptive attributes and may have links to other objects. The attributes may be single- or multi-valued, may be other objects (nested object structure), or may have their values derived from other objects. Derived attributes may either have their values automatically computed when the objects from which they are derived are modified or have the users employ their applications of choice to adapt to these modifications. An important addition to the object model are control attributes. These attributes are attached to objects for the specific purpose of enhancing concurrency and cooperation.

  Being object-oriented with the aforementioned characteristics makes the data model powerful enough to describe the complex data that often dominate product development environments and provide the basis for cooperation support.

- **A flexible model of concurrency control**

  The model allows users and their applications to reveal intermediate results without compromising consistency. It also promotes user mediated consis-

tency (for example, users are notified of changes to objects in which they might be interested, they could dynamically define consistency requirements and negotiate to resolve conflicts). In addition, the model also supports different levels of intra- and inter-group cooperation.

This increased concurrency and cooperation, among individuals and among groups, can increase productivity, reduce product turnaround time, and, equally important, support concurrent engineering methodologies [63] by involving multiple disciplines throughout the entire development process.

- **A framework for data sharing in cooperative product development environments**

  Our approach is to augment both the applications and the database management system with the functionality needed to support cooperation. The framework includes **agents** and a **cooperative database management system**. Each application is encapsulated into an agent which provides the local context for that application. This context is modified both internally by the application itself and externally as a result of changes to relevant objects in the database by other agents. Agents access the database through the cooperative database management system. The cooperative database management system provides, among other features, a dynamic workspace hierarchy for tentative updates and a set of mechanisms to facilitate user mediated consistency and to allow users to track work progress.

1.4 Outline of Dissertation

The remainder of this dissertation consists of chapters 2 through 8.

Chapter 2: Background – presents a walk through computer supported cooperative work. It defines fundamental concepts such as CSCW and groupware. It identifies the key elements of CSCW systems and explains how can CSCW systems be classified based on these, as well as other, elements. The chapter also presents a brief account for the evolution of product development process from the conventional sequential approach to the cooperative concurrent engineering approach and from the use of files to represent and share data to the use of databases. Fundamental work done to enhance concurrent database access in cooperative environments is also included in this chapter.

Chapter 3: Toward a Computer Supported Cooperative Environment – introduces an abstract model of interaction. This model is the setting upon which our work, in the rest of this dissertation, is based. The chapter motivates our research by describing the characteristics of conventional applications and database management systems in a product development environment; these characteristics limit the amount of concurrency which can exist in the conventional environment. The chapter also discusses features which are needed in order to support cooperative work, but which conventional environments lack. Finally, the chapter proposes a framework to provide the needed features and gives a high level view of the framework.

Chapter 4: The Object Model – defines the object-oriented data model used for the representation of data. The chapter presents an overview of the object-oriented approach to data modeling. It describes the different types of objects involved, the relationships that could exist among objects, and the different operations on objects. The object model provides the foundation for later chapters.

Chapter 5: The Cooperative Database Management System – presents the architecture of the cooperative database management system (or Co-DBMS), describes what functionality it adds to an object-oriented data store in order to overcome the weaknesses discussed in Chapter 3, presents the programmatic interface between agents and the Co-DBMS, and summarizes the rules maintained by the Co-DBMS.

Chapter 6: Agents – presents the architecture of an agent, describes what functionality it adds to an application through a set of software modules termed the **application object manager (or AOM)**. The chapter also presents the interface between an application and the AOM, and summarizes the rules maintained by the AOM.

Chapter 7: Cooperative Applications – identifies what is required of an application for it to participate in the system. An application that satisfies those requirements is termed **cooperative application (or co-application)**. The chapter also elaborates what minimal alterations are needed to upgrade an existing application to a co-application, and discusses various levels of cooperation attainable through the coordination of the co-application with the AOM.

Chapter 8: Conclusions and Future Work – presents a final assessment, the significance of this work, and future directions of our research.

Chapter 2

Background

Modern Civilization is entering a new phase, accompanied by a shift from the paradigms of an industrial society to the paradigms of an information society. In this new phase, the axiom that *"information is power"* and should therefore be doled out with extreme caution is replaced with the new axiom that *"information sharing is power"* and everyone should therefore have access to the information they need to perform their jobs. This emanates from the simple reality that, to be competitive in today's global economy, it will take the cooperative efforts of people with different skills to create innovative solutions and innovative products.

Today, the success of most projects relies on the cooperative activities of people. This requires that people communicate, jointly coordinate their activities, and share information and ideas more than ever. The focus of computing in the new information society is on groups, not just individuals. Consequently, any mechanisms or policies to adopt should enable people to work together transcending boundaries of time, space, and functional organization [13].

Computer Supported Cooperative Work (or CSCW) has recently been

established as the field that focuses on the role of computers to support cooperative work. Researchers and developers, in this field, make use of advances in the enabling technologies; mainly portable computing, user-interfaces, and computer networking, to connect disparate information systems, link products with one another, and promote inter-person communication.

CSCW promises major positive impact on many application domains. One such domain is **product development**. Evidently, effective cooperation among members of an interdisciplinary product development group is the key to success. This is because the demand for more and more complex products that exploit technological advances is making it extremely difficult, if not impossible, to assign the responsibility of generating these products to one person or even a group of people who are isolated from one another. Instead, people should be empowered to work both concurrently and cooperatively to pursue their common goal. CSCW provides the needed computer support. Ellis et al. in [16] give useful insights into cooperative computer-based activities:

- concurrent work occurs naturally and spontaneously when the restriction that only one person can access a document at any given time is removed;
- concurrent work can be confusing at times, but conflicts are surprisingly infrequent;
- learning the strategies of, and acquiring knowledge from, other group members is a natural consequence of concurrent, cooperative activities;
- members of a group become familiar with more aspects of the result when they work cooperatively, than if they had worked independently on well-partitioned tasks;

- the fact that many people, having diverse skills, participate to achieve a common shared goal tends to improve the overall quality of the result.

Unfortunately, while cooperative work has been acknowledged as an effective approach to product development, its wide scale adoption has been impeded by the insular approach to data sharing that plagues existing applications and database management systems, (see Chapters 1 and 3 for details). Consequently, a new approach is required to achieve the needed concurrency and cooperation for effective product development. Hence, our work in this dissertation.

This chapter provides the background. Here, we introduce fundamental concepts relevant to computer supported cooperative work, cooperative product development, and cooperative data sharing. We also review research efforts, pertinent to data sharing, which we view as significant contributions toward the realization of cooperative environments.

2.1 Computer Supported Cooperative Work

In recent years, there has been a tremendous surge of interest in providing computer support for many kinds of cooperative work activities. The phrase *computer-supported cooperative work* was coined by Greif and Cashman [26] in 1984 as:

> "Computer-assisted coordinated activity such as problem solving and communication carried out by a group of collaborating individuals."

CSCW involves contributions from a variety of disciplines. In CSCW community, input comes from social scientists attempting to expand our understanding of the requirements that group processes and interactions impose on applications and to evaluate the impact of technology on group performance, computer scientists and electrical engineers exploring new concepts and facilities for developing

computer and communication applications, application builders aiming at creating useful tools for group work, and practitioners trying to combine the diverse systems, applications, and knowledge about work groups to determine how changes can be made to the ways groups work so that future group work is more productive. This cross fertilization has made the field a vibrant one.

CSCW applications are commonly known as **groupware** [34, 29, 4]. The term *groupware* was coined by Peter and Trudy Johnson-Lenz [36] in 1982 as follows:

> "GROUPWARE = intentional GROUP processes and procedures to achieve specific purposes + softWARE applications designed to support and facilitate the group's work."

Groupware is distinguished from normal software by the basic assumption it makes: groupware makes the user *aware* that he/she is part of a group, while most other software seeks to hide and protect users from each other. Groupware is software that accentuates the multiple user environment, coordinating and orchestrating things so that users can "see" each other, yet do not conflict with each other.

CSCW and groupware mark a paradigm shift for computer science, one in which *human-human* rather than human-machine coordination, communication and problem solving are emphasized. This paradigm shift has resulted from a number of converging phenomena:

- the desire to extend personal computing technology to support group interaction and computing, sometimes known as *workgroup computing*;
- the technological opportunities afforded by pervasive computer networking, which has led to widespread use of electronic mail and computer conferencing;

- The merging of computing and telecommunications, and the search for new multi-media communication applications that usefully consume significant bandwidth.

This section identifies fundamental elements of a CSCW environment, proposes a framework for classifying CSCW systems, and highlights several research issues relating to aspects of cooperative work.

2.1.1 Elements of a CSCW environment

As we begin to focus on CSCW environments, we must address the three key areas of *information sharing, communication*, and *coordination*, in conjunction with the *group* and its *activities*. We assert that:

> *Effective cooperation support entails the support for information sharing, coordination of activities, and communication in group, rather than individual, context.*

The group and its activities

Members of a group participating in a given project often engage in a continuous cycle of planning, implementing, monitoring, and modification activities vital to the success of the project.

An integrated multi-perspective environment should evolve to encompass the various private perspectives (personal), the various shared perspectives (sub-group) and the public perspective (group or organization) involved in accomplishing the multitude of group activities.

Development of applications and the way they are used by group members must change to support an integrated multi-perspective approach required for "group operation". Access to applications and services should be facilitated in a transparent manner across the organization. To achieve this end, the integration of existing applications and the development of new applications within an integrated framework are essential. Both existing applications and new applications must be wrapped and/or encapsulated into a federated, heterogeneous integration framework where applications are no longer associated directly with an individual or discipline but at the service of group members scattered across the computing network. In this integrated network, mechanisms should be provided to describe what services are available to users and in what form.

Facilities for information sharing

The functionality to support cooperative work should enable members of a group to cooperatively share information. This means that some information that would have remained implicit throughout an individual project must become explicit so that it can be communicated to other members of the group. Repositories of information should be provided for private, shared, and public use.

Traditionally, each application produced and worked with its own data held in the application's specific format in disk files that are controlled by the application. Consequently, information generated by a group is stored in heterogeneous data formats and in various legacy databases scattered across the organization. Integration of applications of the same class are promoting the creation of database systems that support the operation of applications within their class. Further developments must provide a broader integration in which a network of databases can

support inter-operability between heterogeneous systems. Part of the process requires developing common data representations and standardization of the variety of data exchange and data modeling supported by applications in the cooperative environments.

Current information systems, database systems in particular, must also undergo some changes. The emphasis of current database technology is to keep people from inadvertently corrupting data rather the have a workgroup build something together. As an example consider two designers working with a CAD database. Seldom are they able to simultaneously modify different parts of the same object at the same time and be aware of each other's changes; rather they must check the object out then back in and tell each other what they have done. Many tasks require an even finer granularity of sharing. What is needed is a shared environment that unobtrusively offers up-to-date group context and explicit notification of each user's actions when appropriate.

Facilities for Coordination

In addition to information sharing, members of a work group must also coordinate their joint activities. Coordination refers to the functionality needed for the group work to progress towards mutually agreed upon goals. Coordination is critical for effective functioning of multi-perspective groups. These groups must influence each other so that high quality product is produced within a short turnaround. The major concern here is how to coordinate group activities and resolve conflicts between participants' simultaneous operations such that the coordination overhead does not burden the group and dampen its effectiveness. CSCW demands a fresh approach to control which is specifically tailored for cooperative

work.

In conventional environments, coordination and maintenance of the "current state" of the product is done by the project leader using a virtual workspace that may be composed of paper files, computer archives, tools for project management and so on. Increase in the use of computers and the addition of the "computer supported group work" dimension to the conventional environment adds another dimension to the need of a virtual common workspace to maintain and manage the "current state" of the product. This virtual common workspace must be accessible to all group members, thus providing common visibility of activities and data. This workspace can be the place used by group members to negotiate and reach consensus about their design decisions. It can also be the place used for planning and scheduling of activities, notifying other group members of changes, managing constraints across multiple perspectives, and other coordination and project management activities.

Another important requirement needed for efficient coordination of activities is organization history management. For example, in a design project, it is desirable to capture the design intent and evolution of a product from conceptual design to retirement. Corporate history is useful for designing future products and documenting existing ones. Indexing, linking, and storing various types of documents, and archiving decisions reached in meetings among group members are some of the problems that need to be addressed in this context.

Facilities for communication

The requirements to cooperatively share information and coordinate activities imply that group members must communicate with each other. Communication refers to the functionality needed to support exchange of information among members of a group. We envisage that computer mediated communication would achieve a great deal of success when it derives most of its character from the ways in which people interact (e.g., face-to-face interaction, mail, etc.).

Transition to an integrated multi-discipline environment calls for several changes in the flow of data and information exchanged between applications and among group members:

- an increase in the bandwidth of communication between applications, among group members, and between applications and group members;
- an increase in the degree of "automation" of data and information exchanged;
- a change in the granularity, type and format of data being exchanged.

In conventional environments most of the information exchange takes place face-to-face among users employing traditional computer utilities like electronic mail. Communication and sharing of data between applications is minimal. There is a need for facilities that support data sharing and communication between applications and higher level interaction between group members.

2.1.2 Classification of CSCW systems

A wide variety of CSCW systems have been developed reflecting the many different views of cooperation. The potential benefits of CSCW systems is better understood in a framework for classifying these systems. The most widely used classification of CSCW systems distinguishes them in terms of their abilities to bridge time and

to bridge space [16, 66]. This can be a useful aid in quickly categorizing and later recalling applications, but it has limitations and many researchers have extended it. For example, Nunamaker et al. [59] elaborate it by asking whether "different places" represent different individuals or whole sub-groups. Grudin [30] introduces yet another useful refinement addressing the overly diverse different time, different place activities. Rather than the traditional 2x2 grid, Grudin defines a 3x3 grid to differentiate activities that occur at different but predictable times and places, and different unpredictable times and spaces. Noting the interdependencies among activities, Johansen [35] calls for "any time, any place" support.

Other approaches to classifying CSCW systems are described in [16, 66, 13, 46, 30]. Ellis et al. [16] and Rodden [66] presented taxonomies of CSCW systems based upon application-level functionality. They basically categorized CSCW systems into message systems, conferencing systems, meeting rooms, co-authoring and augmentation, and coordination systems. Dyson [13] classified CSCW systems in terms of managing the work process or the work content, and in terms of centering the control with the users, with a centralized work agent, or with the work itself. Kydd et al. [46] examined the behavior of CSCW systems based upon their predicted ability to reduce the uncertainty and/or resolve equivocality that occurs during group work. Grudin [30] took a broad-based view of CSCW. He suggested that rather than thinking of CSCW as a discipline or a convergence of disciplines, it is more profitably viewed as a forum to which researchers and developers come to exchange ideas. Grudin describes six contexts from which researchers and developers come: activity, group, organizational, technological, research/development, and social.

In this section, we present a framework for classifying CSCW systems based

on the five key elements of group work: activity, group context, communication, coordination, and information sharing. The parameters selected for each of these elements comprise, what we view as, their distinguishing characteristics pertinent to group work.

Activity

- *Scope:*

  the scope of the activity being examined can range considerably; it can focus on a broad application domain, such as product development, education, or banking. A more restricted focus can cut across such domains, such as meeting management, decision support; further refinements are exemplified by the examination of different kinds of meetings and activities within them [53].

- *Structure:*

  activities involved in solving creative problems, such as those tackled by brain-storming, are usually unstructured; on the other hand, prespecified tasks often impose specific structure on their respective activities.

Group context

- *Size:*

  groups can range from two co-authors working together on a paper, to the hundreds of thousands of subscribers of a particular newsgroup. Nunamaker et al. [59] note that meeting dynamics and support differ when the number of participants reaches about 7.

- *Purpose and duration:*

  a group can be organized around a specific narrowly-defined task, such as

writing a document, or can be organized as a team, a project, or an organization; these correlate with another variable; the group lifespan.

- *Homogeneity:*

  Sorgaad [69] identified group homogeneity as a key parameter; groups may consist of peers, such as a group of software engineers; alternatively, a group can span vertical levels of management. such that all of the people in an institution who sign off on employment authorizations; groups can be horizontally mixed, as when support is developed for a newspaper team consisting of reports, editors, proofreaders, and administrators.

- *Cohesiveness:*

  group interactions vary substantially in the degree they are marked to conflict or by shared purpose and agreement; even members of a professionally homogeneous group may have collisions over resources or positions.

- *Structure:*

  management styles vary widely; a simple, hierarchical structure can govern a production group, a consensus, facilitated style can govern a task force, a newsgroup may go entirely unmanaged.

Communication

- *The form of interaction:*

  CSCW systems can be conceived to enhance communication within synchronous interactions, where people interact in real time, or asynchronous interactions, where members contribute at different times; creative problems require group members to cooperate synchronously since the creative input of each group member is required to generate a strategy for tackling the task; in contrast, prescriptive tasks have a previously formulated solution strategy

where group members take on particular roles and work in an asynchronous manner often without the presence of other group members.

- *The geographical nature of interaction:*
  CSCW systems can be conceived to help a face-to-face group, or a group that is distributed over many locations; using this classification, CSCW systems are either remote or co-located. This division is much logical as physical and is concerned with the accessibility of users to each other rather than their absolute physical proximity.

Coordination

The control mechanism within a CSCW system is an additional means of classification which highlights the level of automation each CSCW system provides. The degree of freedom allowed by each type of system provides depth to the classification discussed thus far. A significant area of research in CSCW systems hinges on the amount and form of control CSCW systems provide. Two predominant control mechanisms have emerged: conversation-based control and procedure-based control.

- *Conversation-based control:*
  this is based on the observation that people coordinate their activities via their conversation [77]; the underlying theoretical basis for many systems embracing the conversation model is speech act theory which has developed from the linguistic work of Austin [3], and considers language as a series of actions; for example, The Coordinator [77] is based on a set of speech acts (i.e., requests, promises, etc.) and contains a model of legal conversational moves (e.g., a request has to be issued before a promise can be made); as users make conversational moves, typically through electronic mail, the systems

tracks their requests and commitments.

- *procedure-based control*:

  1. *agent-centered*:

     a user builds his own agent – something as simple as a macro or some calendar rules, or as complex as an expert system to execute rules he/she devices for interacting with other group members and data; the system he/she designs sees him/her as the center, and everything else as the outside world; he/she receives data and requests (commands) from the outside, and sends data, responses and requests back; tasks are usually modeled using AI modeling techniques and an inference engine is used to generate and execute task plans.

  2. *object-centered*:

     where coordination knowledge is stored centrally and often routed by means of forms; the archetype here is the document (or the form) that knows how to mail itself, display itself, update itself from other sources; here, the users write instructions that follow the work around; the object may even send itself out of the system and rely on someone to send it back; the problem is the closure: what happens if the document wonders around and gets lost? who tracks it down? this approach does not offer a high level of representation of the cycle of work to be completed, but depends instead on a model in the user's or programmer's mind; validation of work completion depends on the users rather than the system.

  3. *Process-centered*:

     concentrates on the representation of concurrency as a means of de-

scribing systems; process centered sees the work domain as a whole, and manages work from end to end as a single, complex activity from a central vantage (virtual or physical); its model of the domain includes users, data and applications, the cycle of work and the state of the activity; if user-centered has a user agent and object centered has object agents, then process centered is closer to a group agent, working on behalf of the entire group; the distinction between object-centered and process-centered is subtle: one focuses on the work steps, and the other on the work cycle.

Information sharing

The shared workspace identifies the way in which information is shared and constitutes another means to classifying CSCW systems. Users could cooperate through shared storage, shared application, or messages passing.

- *Shared storage*:
  users interact by sharing data stored in, for example, shared memory, network files systems, and database systems.

- *Shared applications*:
  users interact with the same application program at the same time; this is generally carried out either by providing additional facilities that would effectively convert a single-user application (collaboration transparent software) into one that can be used by a group of remote users, or by constructing new applications that can interact with multiple users simultaneously.

- *Message passing*:
  CSCW systems utilizing message passing are often termed " structured" or "active" message systems and assume an asynchronous and remote mode

of cooperation; the assumption underlying these systems is that members of a group cooperate by exchanging messages; these systems are based on the principle of extending the a mount of machine processible semantic information available by adding syntactic structure to the existing message structures.

2.2 Product Development

This section presents the evolution of the product development process from the conventional sequential approach to the more advanced concurrent engineering approach that promotes cooperation among product development groups.

2.2.1 The conventional approach

A product development process, following the conventional approach, is comprised of a sequence of phases starting with marketing studies for the need of a new product, the identification of requirements and the development of the specifications, followed by several phases in which the product is gradually defined. At the end, a product is manufactured, placed in service, and maintained [48]. Earlier design decisions may limit the range of design decisions which are possible in the final phases. Feedback from the effect of new design decisions are propagated upstream, and previous design decisions may be revised. The conventional product development process is *sequential* but includes a set of iterative cycles.

It has been indicated that much interaction between different product developers with different specializations takes place between phases in the product development process [8]. Product developers from different specializations interact, cooperate, negotiate, and commit design decisions in each of the product develop-

ment phases. At each stage work by different perspectives is synchronized, reaching consistency among perspectives, and then moving on to a new phase in the product development process. Work may proceed for long periods of time where inconsistency between different disciplines may prevail.

The conventional approach to the development of applications is to support the single-specialization product development activities. Powerful applications are being realized to address well-structured problems with well-understood theoretical frameworks within a given area of specialization. Computers are helping the individual, but they may be complicating the work of the group. Computers promote the distributed way of working but they still do not provide support for the basic set of operations required by a group of cooperating product developers: *human-human communication, human-assisted activity coordination*, and *cooperative data sharing.*

2.2.2 The concurrent engineering approach

A new design methodology is gaining acceptance within industry, government, and academia. This methodology is known as **concurrent engineering (or CE)**. The commonly accepted definition of CE was published by the Institute for Defense Analysis [76], and is stated as follows:

> "CE is a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support. This approach is intended to cause the developers, from the outset, to consider all elements of the product life cycle from conception through disposal..."

The ideas of CE have been around for many years, but because they stand in contrast to the current practice of sequential product development, CE is gaining momentum as part of the strategy to meet the demand from competitive international markets for the development of more complex products of higher qualities in shorter times [64].

CE implies a significant change in the ways products are developed and sustained. In conventional sequential development practices, information flows one way: from design to manufacturing. It is a cyclic process, each phase goes through one or more re-design and test cycles to account for the effect of process on the design. CE, on the other hand, promotes a dynamic, interactive feed forward of the knowledge gained and created during the product development process. In this approach, specification changes and new requirements are propagated downstream by *providing simultaneous access to the current design state to all product developers who contribute with design decisions during the product development life cycle*; conflicts in manufacturing and logistics perspectives are propagated upstream, similarly.

CE promotes freer and richer interchange of information between a group of product developers who can contribute to making a better and cheaper product in a shorter time. One approach to promote this group organization is to develop a computing environment that facilitates cooperation and concurrency of activities among the product developers conforming the group. We call this environment, a **cooperative product development environment**, and the development of such an environment is the target of our research. In this dissertation, in particular, we address data sharing issues relevant to cooperative product development environments.

2.3 Data Sharing in Product Development Environments

Effective sharing of data is central to cooperation. The representation model of data is a determining factor in realizing such effectiveness [49, 50]. This section describes the evolution of product development environments in recent years from the use of data files to the use of databases.

2.3.1 Data files

Most existing product development applications were developed by different vendors with different goals, and before the importance of inter-operability was recognized. For this reason, emphases were placed on the functionality of that one application, that is, on the manipulations of data which the application would perform. The fact that other applications might perform manipulations on related aspects of the product, or even that other application exist, was not initially considered.

The applications that resulted from this insular philosophy have their own private repositories of data [48]. These repositories are collection of files. The semantics of the contents of these files are unknown to all but the one application which uses those files and for which the file format was developed. Thus, inter-relationships among the data sets of different applications, which may represent multiple aspects of the same product, are ignored and it is impossible to automatically maintain consistency among their views. Instead correspondence among various files must be manually maintained. Doing so in a setting of concurrent development, that is, involving a number of product developers, is a complex, time-consuming, and error-prone task.

Many vendors have recently made public the formats of files used by their applications. This openness has motivated the creation of a new market, that of **"application integration"**, in which translation utilities or filters are developed to convert from one vendor's file format to another's.

Absence of communication among application vendors during development of applications has resulted in a large number of file formats. In order to reduce the number of file formats in use and to encourage the creation of filters, various standards committees are actively defining standard file formats which implement common views. The advantage of having applications which use standard file formats along with the filters to translate among the various formats is that an application will not require modifications to be used collectively with other applications and therefore the investments in existing application suites are preserved.

The application integration approach is indeed a positive step toward interoperability and thus sharing of efforts among a group of product developers. A major problem, however, exists that prevents the acceptance of the approach as a universal solution to share data in product development environments: that of the coarse granularity of change, namely at the level of an entire file. Limiting the granule size at the level of files inhibits support for performing incremental analyses on the evolving product. Change notifications to interested parties are also restricted to a coarse level of detail; that is a file. Furthermore, since concurrent updates to different parts of the same file by two or more product developers will result in inconsistencies, and the unit used is the file, two or more activities can proceed concurrently only to the extent that they involve different unrelated files.

2.3.2 Databases

In the data-file approach to data representation in product development environments, emphases were placed on the use of a particular application at a particular time and on translating data into a format suitable for that application. The usefulness of these applications is thus limited, because the data they manipulate are not integrated. The data management needs of the product development environment are extensive and complex [50, 68]. The need in the product development environment for capabilities which traditionally have been associated with a database management system, such as structured information, an integrated data model, access control, and concurrency control, has become apparent in the past few years [48, 27].

Placing data in a database makes them available for use by many applications and product developers. The database provides the same programmatic interface and integrated data model to all applications. Applications read and update the data in the database, and during their operation cache their own views of those data; such a view enables the application to efficiently perform its task. Each application derives the view it needs from the integrated data model offered by the database. Conversely, when an application needs to induce change in the database, it must first translate updates from its view to the integrated data model before submitting them to the database. Thus, in the database approach, there exist filters, similar to those used in the data-file approach, to translate from the data model offered by the database to and from the view employed by the application. A filter is application-dependent and is developed by the application vendor rather than by an application integrator, and is thus part of the application.

A database offers several advantages over the use of data files to store data

used in product development:

- the integrated data model of the database is advertised; any application vendor is free to develop applications which adhere to that model;
- a database accepts incremental updates: thus, an application that updates a portion of a product need not re-enter the entire product; instead it can submit only those updates which represent the delta of change effected by the application on the product;
- the database serves as central point where access control specifications can be stored;
- a database management system typically includes techniques to ensure high availability of data in the event of hardware failures, and the ability to roll-back to previous states or undo recent changes.

The use of existing database management systems, however, does not go without problems. One major problem emanate from the methods used to control concurrent access to shared data. In Chapter 1, we discussed some characteristics of the conventional approach to concurrency control, employed in existing applications and database management systems, which severely restrict cooperation. Further examination of these characteristics, as well as others, is presented in Chapter 3. The next section reviews some recent research efforts aiming at enhancing concurrency and promoting cooperation.

2.4 Concurrency Control Research

Recently, vigorous research has been conducted to overcome the limitations of the conventional approach to concurrency control. In this section, we present seven

recent research studies dealing with problems closely related to our work. The first five studies suggest the use of extended transaction models for long-running cooperative activities, the sixth study deals with coordinating change to a set of files in a software development environment, while the seventh, and last, study concentrates on real-time group text editing.

2.4.1 Split-transactions, commit-serializability, and participation domains

Split-transactions were proposed by Pu et al. in [62]. They were proposed mainly for supporting open-ended activities. These activities are characterized by (1) uncertain duration, (2) uncertain developments (actions cannot be foreseen at the beginning), and (3) dependency on other concurrent activities. Pu et al. define a notion of consistency called **commit-serializability**. The basic idea of commit-serializability is that all sets of database actions included in a set of concurrent transactions are performed in a schedule that are serializable when the actions are committed. The schedule, however, may include new transactions that result form splitting (or joining) the original transactions. Splitting a transaction divides an ongoing transaction into two or more serializable transactions by dividing the actions and the resources between the new transactions. The resulting transactions can proceed independently from that point. Also, these transactions behave as if they had been independent all along while the original transaction disappears entirely as if it had never existed. Thus splitting a transaction can be applied only when it is possible to generate serializable transactions.

The main purpose of split transactions is to commit one of the split transactions and release useful results from the original transactions. The other split transaction continues. Three advantages accrue: (1) dynamic restructuring of

transactions: users are allowed to restructure their long transactions dynamically; (2) adaptive recovery: committing part of the work done by a transaction which then will not be affected by subsequent failures; and (3) reducing isolation: releasing resources by committing part of a transaction.

The split and join operations do not support interaction between concurrent activities, if used solely. For this reason, Kaiser in [37] combined these operations with the notion of **participation domains**. A participation domain defines a group of transactions as participants in a specific domain. A transaction is placed in a domain in order to share partial results with other transactions in the same domain in a non-serializable manner, but it must be serializable with respect to all transactions not in the domain.

2.4.2 Proclamation-based concurrency control

Jagadish and Shmeuli in [33] presented a transaction model which aimed at providing a framework for transactions to cooperate without sacrificing serializability as a notion of correctness. Cooperation typically requires one transaction relying on certain behavior by another transaction. Jagadish and Shmeuli stated that, while this reliance is usually based on some higher level knowledge, it can often be reduced to a reliance on a particular update behavior; in particular, a transaction may be able to predict, at least partially, what value it will write for a particular data item X well in advance of the transaction completing its computation and committing; another transaction, wishing to read X, may be able to perform useful computation even if it does not know the exact value of X, but instead merely that X belongs to some *set* of values.

In Jagadish and Shmeuli's model, transactions, as in the conventional model,

are flat, deterministic, and are assumed to transform consistent states into consistent states. Transactions are also monotonic; if each read operation of a transaction is made to read a subset of what it actually reads then each update operation will produce a subset of the values it actually produces.

Transactions cooperate by issuing **proclamations**. A proclamation is an (implicitly or explicitly specified) set of values, one of which the transaction *promises* to write when it commits. So, a proclamation offers incomplete information concerning future database states. A transaction, upon finding unavailable a data item that it wishes to access, may request the current item-holder for a proclamation. The transaction can compute with the incomplete information provided in the proclamation, and can commit after writing conditional multi-values.

Jagadish and Shmeuli provided theoretical basis for the proclamation model and they outlined an implementation strategy, including a lock-based transaction manager and a transaction compiler extension to handle sets of values.

It is to be noted that, if no proclamations are issued, Jagadish and Shmeuli's model degenerates to the conventional flat transaction model based on serializability. Using proclamations, however, enhances concurrency without requiring detailed knowledge of the semantics of the particular application. Extensions of Jagadish and Shmeuli's model to include nested transactions warrant further investigations.

2.4.3 Nested transactions with predicates and versions

Korth and Speegle in [41] presented a formal model that allows mathematical characterization of correctness without serializability. They called the model "**Nested**

Transactions with Predicates and Versions (or NT/PV)". The model combines three features that lead to enhancing concurrency over the serializability-based models: (1) multi-level transactions, (2) explicit consistency predicates, and (3) versions of objects.

The database in Korth and Speegle's model is a collection of entities, each of which has multiple versions (i.e., multiple values). The versions are persistent and not transient like in the traditional multi-version scheme [7]. A specific combination of versions of entities is termed a unique database state. A set of unique database states that involve different versions of the same entities forms one database state. In other words, each database state has multiple versions. The set of all versions that can be generated from a database state is termed the version state of the database. A transaction in Korth and Speegle's model is a mapping from a version state to a unique database state. Thus, a transaction transforms the database from one consistent combination of versions of entities to another. Consistency constraints are specified in terms of pairs of input and output predicates on the state of the database. A predicate which is a logical conjunction of comparisons between entities and constants, can be defined on a set of unique states that satisfy it. Each transaction guarantees that if its input predicate holds when the transaction begins, its output predicate will hold when it terminates. (Compare this with the assumed consistency of conventional transactions.)

A transaction in Korth and Speegle's model is a quadruple (T, P, I, O), where T is the set of subtransactions, P is a partial ordering on these subtransactions, I is the input predicate on all database states, and O is the output predicate. The input and output predicates define three sets of data items related to a transaction: (1) the input set, (2) the update set, and (3) the fixed point set, which is

the set of entities not updated by the transaction. Given this specification, Korth and Speegle define a parent-based execution of a transaction as a relation on the set of subtransactions T that is consistent with the partial order P. The relation encodes dependencies between subtransactions based on their three data sets. This definition allows independent executions on different versions of database states.

Finally, Korth and Speegle defined a new multi-level correctness criteria: An execution is correct if at each level, every subtransaction can access a database state that satisfies its input predicate and the result of all the subtransactions satisfies the output predicate of the parent transaction. But since determining whether an execution is in the class of correct executions is NP complete, Korth and Speegle consider subsets of the set of correct executions that have efficient protocols. (See [41] for more details.)

korth and Speegle's model is not readily applicable in cooperative environments. This is because the input and output predicates of a transaction are defined against the global database state and cannot be tailored to the task at hand.

2.4.4 Cooperative transaction hierarchy

The cooperative transaction hierarchy concept was introduced by Nodine and Zdonik in [58] for supporting cooperative applications like CAD. Serializability in the conventional transaction model restricts cooperation between transactions by not allowing the transactions to exchange information through accessing (i.e., reading and updating) common data. To overcome this problem, Nodine and Zdonik proposed to structure a cooperative application as a rooted tree called **a cooperative transaction hierarchy**. The external nodes of the hierarchy represent the transactions associated with the individual designers. An internal node is

called a transaction group, and contains a set of members (i.e., children) that cooperate to perform a single task. The term cooperative transactions in the model refers to the transactions with the same parent in the transaction tree. Cooperative transactions need not be serializable; instead, the transaction group (i.e., parent) of the cooperative transactions defines a set of rules, denoting patterns and conflicts, that regulate the way the cooperative transactions should interact with each other. Patterns and conflicts are defined in terms of a set of finite-state machines (or FSMs). A FSM specifies, for a set of objects, the operations allowable for each cooperative transaction, and the allowable ways of interleaving the operations of related cooperative transactions.

The main contribution of cooperative transaction hierarchies is the substitution of a notion of user-defined correctness for the notion of of correctness defined by serializability. The notion of user-defined correctness criteria allows different parts of a shared task to use different correctness criteria that are suitable for their own purposes. Because isolation is not required, the cooperative transaction hierarchies allow close cooperation between transactions and also help to alleviate the problems caused by long-lived transactions.

Several extensions of the basic model have been proposed. Skarra [68], instead of using FSM, used a more complex, Turing-complete grammar to define patterns and conflicts in a transaction group. Nodine et al. [57] discussed a model of operation-based recovery in addition to synchronization. Finally, Heiler et al. [32], in addition to the execution of individual requests, added the execution of sub-requests and defined an architecture that exploits the facilities of an Object Management System.

Applying the correctness criteria, in the models above, depends on a recognizer and a conflict detector to enforce semantic patterns and conflicts. The recognizer and the conflict detector must be constructed for each application. The utility of cooperative transaction hierarchies is further limited due to following two assumptions: (1) cooperative transaction hierarchies *mirror* organizational units, or decomposition of the product, or decomposition of the development process; (2) a cooperative transaction hierarchy is determined a priori and is fixed throughout the design process. The work in this dissertation, as will be shown in the following chapters, relaxes these restrictions.

2.4.5 Lazy consistency

Narayanaswamy and Goldman in [56] addressed the problem of resolving global conflicts introduced by local changes in cooperative software development. The aim of their work was to identify the technical basis to support such resolutions. Narayanaswamy and Goldman stated that, in cooperative software development, the basis should be a network wide notification of **proposed changes**, rather than actual changes to objects.

The proposed change notifications happen within the context of a larger transactional unit called an **evolution step**, which corresponds to a single goal of the programming team. Dependencies between objects are used to define who has a stake in each proposed change. Support is provided for affected programmers to approve or reject each proposed change. It follows that, within the context of an evolution step, programmers can *explicitly* state when the system is expected to be in a consistent state, and when it is tolerable for it to be in an inconsistent state.

The causal relationships between proposed changes are maintained so that pro-

grammers' negotiations can be supported. Using these concepts, the authors defined a notion of consistency called **lazy consistency** which supports a process of *gradually* making each evolution step internally consistent and consistent with respect to other volatile steps that might be pursued concurrently.

Narayanaswamy and Goldman's model allows a great deal of concurrency within a single evolution step. Work on inter-step consistency, however, is still in progress. It is also worth mentioning that, an evolution step, following Narayanaswamy and Goldman's model, has a flat structure; it represents a single goal with no support for multiple goals or sub-goals.

2.4.6 Coordination consistency

Harrison et al. in [31] presented a formal model of concurrent development, in which development consists of a collection of modification activities that change files, and merges that combine the changes. They defined a weaker than serializability notion of consistency called **coordination consistency** that ensures that changes are not inadvertently destroyed and that the changes of each modification activity are correctly propagated to subsequent modification activities.

In Harrison et al.'s model, an artifact is represented by a set of files kept in a master store. Development consists of modification activities and merges. A modification activity is a set of changes, made in isolation in a separate store. Multiple modification activities can occur concurrently, each in its own store. For the the set of changes made during a modification activity to become visible outside its store, that store must be merged with other stores. Ultimately, all changes that are to become part of the artifact must be merged into the master store. The basic aspect of coordination consistency is ensuring that the developing artifact remains

consistent in the face of concurrent modifications without reference to the details of the artifact.

Harrison et al. based their work on the premise that: during the course of development, much material is examined that can nonetheless be changed without adversely affecting the work in progress. An underlying assumption is that the work of various developers is loosely coupled.

A drawback of Harrison et al.'s approach is their use of files as the granularity of change. As mentioned in Section 2.3.1, the use of files inhibits performing incremental analyses on the evolving product and impedes cooperation. The authors do not mention how merges will be carried out. In addition, their conditions for collisions correspond to those where changes are to the same object in both activities. In our work, as presented in this dissertation, we allow a more open-ended definition of a collision, with applications and people deciding when a collision has arisen.

2.4.7 Operation transformation

Ellis et al. in [14, 16] described an algorithm for ensuring precedence and convergence properties in real-time CSCW systems. No transaction or locking is involved. Instead, operations are transformed when necessary; the algorithm must know some semantics of the operations.

The model assumes data replication at all sites and global operations; an operation executed at one site must be executed at all sites. The proposed concurrency control algorithm is based on the following premise: instead of executing $O_1 o O_2$ at one site and $O_2 o O_1$ at the other, we execute $O_2' o O_1$ and $O_1' o O_2$ where O_1' and

O_2' are transformed operations obtained from the original operations O_1 and O_2 respectively and o is the composition operation. O_1' and O_2' are calculated so that $O_1' o O_2$ when applied to a site object has the same effect as $O_2' o O_1$.

Operation transformation has been used in the GROVE editor [15]. In that context, each user has his/her own copy of the editor, and when an operation is requested, this copy locally performs the operation immediately. It then broadcasts the operation along with a state vector indicating how many operations it has recently processed from other workstations. Each editor copy has its own state vector, with which it compares incoming state vectors. If the received and local state vectors are equal, the broadcast operation is executed as requested; otherwise it is transformed before execution. The specific transformation is naturally dependent on the operation type (e.g., an insert or a delete) and on the log of operations already performed.

The assumptions of full data and application replication and the use of only transformable global operations restrict the applicability of Ellis et al.'s algorithm to specific application domains which can exhibit this kind of behavior and which require tightly coupled cooperation among users. If such application domains exist, then employing Ellis et al.'s algorithm could enhance their responsiveness.

2.4.8 Remarks

We presented several new approaches that address the differences between concurrency control requirements in cooperative environments and conventional data processing environments. Surveys of many other approaches exist in [28, 6, 17]. Although all of the approaches presented in [28, 6, 17] and this dissertation fulfill at least one of the concurrency control requirements, none of them provides ad-

equate support for all requirements. Many of the approaches, especially those in [28], have a relatively narrow, domain-specific scope. Moreover, the technical support for communications and concurrency control, especially in approaches that achieve higher levels of concurrency and cooperation, is more often tightly integrated into the domain-specific functions of the system. The framework described in this dissertation, in contrast, is intended to provide mechanisms that render a more general and encompassing solution. Our work, in addition, addresses several important issues that are, so far, barely addressed by the multitude of existing models of data sharing managers in cooperative environments. These issues include:

- the interface to the applications;
- the interface to the underlying DBMS;
- active participation of the system in handling notifications;
- access to the status of work in progress.

The following chapters describe our work.

Chapter 3

Toward a Computer Supported Cooperative Environment

This chapter motivates our research by describing the characteristics of conventional applications and database systems in a product development environment; these characteristics limit the amount of concurrency which can exist in the conventional environment. The chapter also discusses features which are needed in order to support cooperative work, but which conventional environments lack.

Section 3.1 introduces an abstract model of interaction. That model is the setting upon which our work, in the rest of this dissertation, is based. Sections 3.2 and 3.3 discuss the operation of conventional applications and database systems in a product development environment. Section 3.4 shows the limitations of the conventional environment that render it inadequate for cooperative work. Next, Section 3.5 discusses features of a cooperative product development environment which compensate for weaknesses of the conventional approach. Finally, Section 3.6 proposes a framework to provide the features discussed in Section 3.5 and gives a high level view of the framework. Later chapters present in detail what

data model is used by the framework, what services the framework provides, how the components of the framework operate, what rules are adopted by the various components, and what levels of consistency are guaranteed.

3.1 High Level System Model

At the highest level, an environment for product development consists of **product developers** employing **applications** to access and manipulate data stored by the **database management system**. See Figure 3.1. Each of these components is discussed below.

3.1.1 Product developers

A product developer is a human who is involved in the development of one or more products. Product developers may assume different **roles** throughout the development process. A developer's role determines his or her right to update specific objects in the database.

3.1.2 Applications

An application consists of application code, internal state, and a translator from and to the data model offered by the database. Applications are used by product developers to access and manipulate objects in the DBMS. Applications provide a user interface to the users who use them. The DBMS can be used only indirectly through applications. Applications are independent. They jointly access the DBMS, but each application is unaware of the existence of other applications. A product developer can employ multiple applications simultaneously.

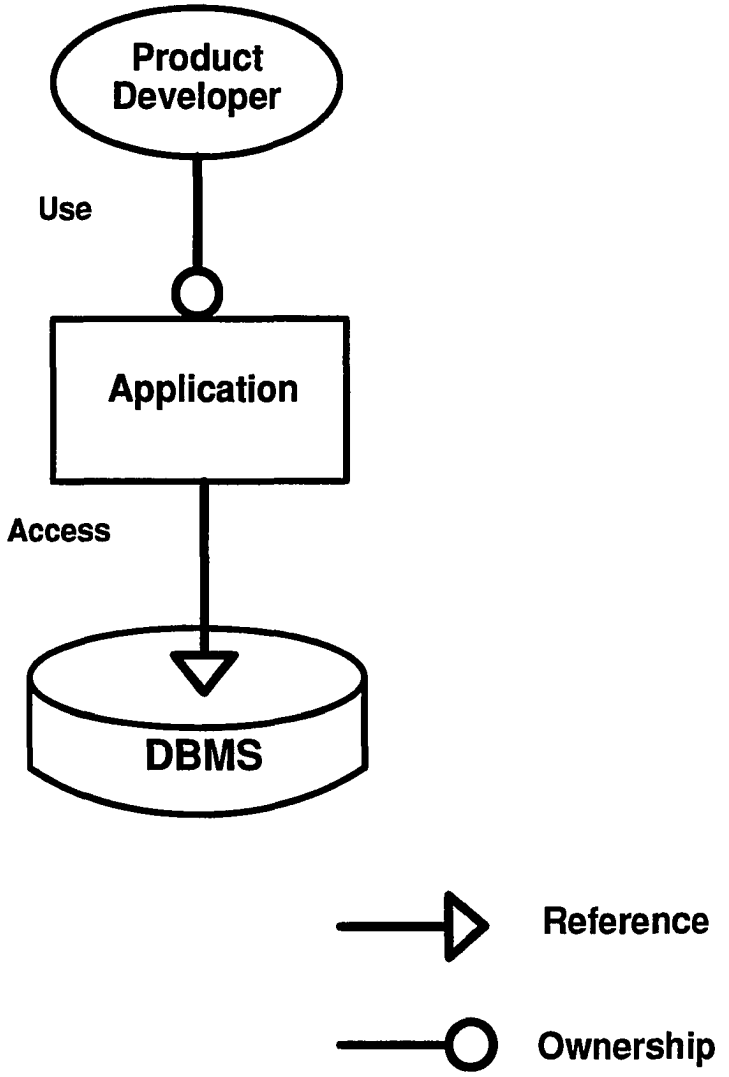

Figure 3.1: High Level Interaction Model

3.1.3 The database management system

The DBMS offers applications the ability to access and make persistent changes to data stored in an object store. Since applications, rather than the DBMS, offer a user interface for product developers, then the DBMS needs to provide only a programmatic interface to the stored data for use by applications. It is to be noted that the DBMS need not be physically centralized, the use of "the DBMS" is not meant to exclude multi-DBMSs [18] or distributed DBMSs [11], but rather to refer to the aggregate functionality of the database system being used.

Data are stored in an **object store**. The DBMS has work areas called **workspaces**, in which tentative updates are made. When those updates are no longer tentative, they are **committed** to the object store. Before an application can update an object in a workspace, the application must **check-out** that object into that workspace. This action indicates an intent of the application to update that object in the workspace. Intent to update an object is released when the object is **checked-in**.

3.2 Conventional Applications

The operation on data of a conventional application consists of recursions of the following five steps.

- **Reserve**

  Before an application can manipulate database objects, it must secure write locks on the objects to be updated, and read locks on objects that will be used during the course of its operation.

- **Load**

  An application utilizes data structures particularly defined for its efficient operation. Since different applications perform different tasks, the data structures selected to maximize that efficiency are application dependent. The database offers one integrated data model for applications; data must be accessed and stored using this data model. No single data model can efficiently support the multitude of representations required by different applications [10].

  Applications interact with the database using the common data model, as defined by the database schema. If the database offers an object data model, then the contents of the application's data structures are derived from the objects read from the database, and the updates from an application must be presented as updates to objects.

  After the appropriate locks have been acquired, an application loads from the database those objects which it needs to access for read or update. If a translation is needed between the view offered by the database and the data structures used by the application, then it is the responsibility of the application to secure this translation.

- **Manipulate**

  After loading the desired data, these data are manipulated (in the appropriate application's format). It may be impossible to predict the duration of this step; data manipulation may extend over a period of several hours, days, maybe even weeks or months. In other words, operations on objects by applications in a product development environment are often long-lived

[5, 6].

- **Unload**

  After the application has manipulated the data to the satisfaction of the person who initiated the application, the internal state of the application is translated to changes to objects in the database and these generated changes are sent to the database. These objects then assume their new state in the database.

- **Release**

  After an application has finished its manipulation of objects, it should release the locks it acquired in the first step, so that other applications can acquire locks on those objects.

3.3 Conventional Database Management Systems

This section presents the concept of workspaces in database management systems used in product development. The reasons for having workspaces are explained. The section also describes how objects are conventionally manipulated in these workspaces.

3.3.1 Workspaces

A conventional product development database contains a **public area** and a set of work areas (or sub-databases) called **workspaces** [10, 54]. Stable products are placed in the public area of the database. All updates to data are encapsulated within workspaces.

Public area

The public area contains the collection of approved data. In engineering terms, approved data means data that have undergone several levels of verification and authorization by some group of people involved in the product development process (e.g., the developer, the group leader, and the project manager). Data that have not reached full approval have to be marked as so, in order that derived objects be also regarded as tentative and subject to the final approval of the data they were derived from. The size of the schema is usually a problem, since it comprises of a very large number of objects. Even if there were no data quality limitations to updating the full database, the sheer size of the schema and the data volume in a large project make it impossible to allow direct updates to the public area other than the integration of final designs [10].

Workspaces

The length of interactive engineering transactions, the different levels of data quality, and the desire to narrow the focus to some subset of objects, each represents a powerful reason to generate workspaces [74].

A workspace is a region in the database which holds *copies* of objects. Applications make changes only to objects in workspaces. These updates are tentative; an application automatically *commits* changes to the public area when the desired state is achieved. Instead of committing changes in a workspace, the changes can be *aborted*, which means that updates since the last commit are discarded and the view offered by the workspace is the same as that offered by the public area.

3.3.2 Updates in workspaces

Copies of objects in the workspaces hold the tentative state of the objects. A workspace offers a *view* of objects, which is the collective state of the objects. The view of the objects offered by a workspace is the view of the objects in the public area modified by some update *delta*; this delta is the concatenation of all updates (modifications, creations, and deletions) in that workspace since the last commit. Each workspace has an associated *transaction log* which records what updates have been performed to objects in the workspace. The transaction log is useful in the event that one or more updates must be undone.

3.3.3 Commit and abort

Let $V_W(t)$ and $V_P(t)$ represent the views offered by workspace W and the public area P at time t, respectively. Let u_i represent the i_{th} update to W, and $\Delta U_W(t) =< u_1, u_2, ..., u_n >$ represent the list of all updates applied to W through time t since the most recent commit at time $t_{prevCommit}$. Then the semantics of update, commit, and abort are as follows.

- Suppose workspace W is created at time $t_{Initial}$, then

$$V_W(t_{Initial}) = V_P(t_{Initial}), \tag{3.1}$$

and

$$\Delta U_W(t_{Initial}) =<>, \tag{3.2}$$

that is to say, the initial state of objects in the workspace is the same as that of the public area at initialization time.

- For all $t \geq t_{Initial}$,

$$V_W(t) = V_P(t) + \Delta U_W(t), \tag{3.3}$$

in other words, the state of objects in the workspace is the same as that in the public area except for updates made to objects in the workspace.

- If update u occurs at time t_u, then

$$\Delta U_W(t_u) = \Delta U_W(t_u - 1) + < u >, \tag{3.4}$$

which is to say that updates have a commulative effect on the workspace and each previous update is a prefix to its successor update.

- Suppose updates to W are committed at time t_{Commit}.

Then, for all $t, t_{prevCommit} \leq t < t_{Commit}$,

$$V_P(t) = V_P(t_{prevCommit}). \tag{3.5}$$

Furthermore,

$$V_W(t_{Commit}) = V_P(t_{Commit}) = V_P(t_{Commit} - 1) + \Delta U_W(t_{Commit} - 1) \tag{3.6}$$

and

$$\Delta U_W(t_{Commit}) = <>, \tag{3.7}$$

in other words, updates in the workspace have no effect on the public area until the updates are committed, and all updates are applied *atomically* at commit time (i.e., either all or none of the updates are involved in a commit).

- Suppose updates to W are aborted at time t_{Abort}, then

$$V_W(t_{Abort}) = V_P(t_{Abort}) = V_P(t_{prevCommit}) \tag{3.8}$$

and

$$\Delta U_W(t_{Abort}) = <>, \tag{3.9}$$

in other words, aborting changes in the workspace causes them to be discarded.

3.3.4 Check-out and check-in

Before an application can read or update an object in a workspace, it must **check-out** that object into that workspace. Check-out is an association among applications, workspace, and object(s). Check-out may be made either for read or update access. An object may be checked-out for update access by only one application at any given time. Furthermore, checking-out an object for update access excludes read access by different applications. Thus in the conventional product development environment, the check-out of an object O for update in workspace W by application A is an *exclusive write-lock* on O given to A. This limits updates to O to occur only in W and only by application A, and checking-out for read access is a shared read-lock.

The act of **check-in** releases the intent to read or update an object which was checked-out. Check-in is the inverse of check-out. An application must apply internal updates to the workspace or abort them before it checks-in objects.

3.4 Limitations of Conventional Environments

As described above, in a conventional environment, the database system offers the protocol of check-out and check-in which ensures that different threads of activity which may be interrelated are not run concurrently, or are scheduled in a way that has the same effect as though the threads' execution times do not overlap – this is called serial schedule [61]. Using this protocol, applications have exclusive access to database objects for the duration of their operation. This is necessary because applications have been built to assume that data in their *read set*, that is, those data upon which it has predicated its operation, are not changed by users external to the application. Allowing other applications to change those data might adversely affect the integrity of the application's results.

But these characteristics are counter to the premise of cooperative product development, in which multiple users use multiple applications to complete the work as a team. Thus, a conventional database and conventional applications are inadequate in a cooperative environment.

3.5 Features of a Cooperative Environment

This section explains those features of a cooperative environment which support cooperative work and which are not offered by the conventional environment.

3.5.1 No exclusive access

As mentioned earlier, when an application checks-out an object for update, the database grants the application exclusive access to that object. This approach to access control has its origin in business transaction processing systems which em-

phasize a large number of short, simple transactions issued on behalf of users who are oblivious of one another. Users requesting the transactions are not allowed to assume that the state will be retained across transactions. By contrast, in product development environments such as computer-aided design or software development, developers or development groups share some concrete, often complex, conceptual artifacts for long periods of time. During the development process, different product developers interact together, and with the database, to achieve their common goals. Conventional techniques to controlling concurrent access in a DBMS are inadequate in a cooperative environment, since a database management system for product development must permit activities of undetermined length which do not have all their operations known a priori and which do not preclude access to data by many other transactions.

Consider, for example, a computer-aided design environment. It is not feasible to have exclusive access to the entire design since many designers work on overlapping aspects of it simultaneously. Even exclusive access to only one portion of a design is also limiting: parts of a design are interrelated, and it may be useful to have two or more applications share updates to the same portion of a design. For example, some designers may wish to share updates to the same portion, or one designer might want to run several applications simultaneously on the same design data; applications must not be constrained to be invoked in a serial fashion. Without exclusive access, there must be other mechanisms which permit applications to maintain views of the design consistent with the database.

3.5.2 Up-to-date knowledge about changes to shared data

Keeping an application informed of the ways in which its read set has changed enables it to adjust its view to match the changing state of the database. Ap-

plications should not be expected to have knowledge of the semantics of other applications, however. Thus a central mechanism is needed which will notify an application when data it has cached are changed by other agents. That mechanism is part of our framework.

3.5.3 Applications adapt to changes

Even with a mechanism that guarantees applications that they are notified of changes to database objects, in order for an application to interact harmoniously with other applications, it must respond to these notifications in a proper fashion. This includes not only making its cache of data consistent with the database, but possibly undoing or making compensating changes to updates it had performed but not yet committed to the database. Exactly what an application does depends upon the semantics of the application and the data. How notifications should, in general, be handled by an application is discussed in Chapters 6 and 7.

3.5.4 Use of differential updates

In a product development environment, most applications, when they execute, make incremental rather than sweeping changes to the product [6, 9]. But if an application submits its updates to the database as "the new state of the product" rather than "the differential changes applied to the product", the incremental information is lost. Incremental information can be lost in a similar fashion when a workspace is committed to the public area.

Incremental information is useful because it enables notifications of changes which are sent to other applications to take the form of a small rather than a large delta. A small delta can more easily be handled by an application. Applications in the cooperative product development environment will update the database by

submitting a list of differential updates in order to preserve knowledge of incremental changes.

3.5.5 Extensibility and integration

Extensibility refers to the ease of incorporating new capabilities (such as new applications) in the environment [72]. Applications must share an open-ended environment which can be extended to accommodate new applications without necessitating change to existing applications or other parts of the environment. Furthermore, it must accommodate sets of applications which are tightly coupled; such as two applications sharing updates to the same objects, as well as loosely coupled; such as the applications under the control of different designers working on different aspects of the design.

Another principal quality sought in the development of cooperative product development environments is integration. Integration refers to consistent interfaces, easy context switching, and efficient communication between applications. Interaction with the environment should be in a uniform way. In addition, applications should share information among themselves, assuring that users are not obliged to supply the same information multiple times, nor needlessly paying for computation of available information. Environment components should be shared whenever possible as well, to keep the size of the environment down, and to prevent performance penalties due to excessive paging and thrashing.

Several investigations have underscored the importance of extensibility and integration in product development environments, however, they have also indicated that there are some fundamental tensions between them: a tightly integrated environment is easiest to achieve if the environment is limited in scope and static

in its content and organization; conversely, broad and dynamic environments are typically loosely coupled and hence impose excessive burdens on users [60, 72]. Consequently, efforts should be directed toward maximizing both extensibility and integration while putting into consideration the trade-off between them.

3.5.6 Multiple levels of cooperation

Product developers use separate workspaces when the objects checked-out into those workspaces are unrelated, or when integration of objects into a parent object is being deferred. At other times, when product developers want to work on very closely related parts of the product, any partitioning may seem artificial and may impose an unacceptable overhead. In this case, product developers should be granted the ability to access the same objects in the same workspace. When product developers share access to some object, the views of the applications employed by the product developers should be kept synchronized with that of the workspace.

It is to be noted that such a constructive utilization of applications is based on the premise that *users sharing access to the same objects are willing to communicate among each other to reconcile their differences and coordinate their conflicting activities.* This concept of cooperation among users is absent in the conventional environments, however, it should be intrinsic to cooperative environments.

3.5.7 Dynamic workspace hierarchy

The notion of a workspace, as presented in the previous section, can be generalized to a hierarchy [38, 1, 20, 54, 74]. Workspace hierarchies support our view that a complete product comes to existence step by step through cooperation. At the top of the hierarchy is the root workspace W_{root}. Every workspace W, except W_{root}, has a superior workspace *Superior(W)*, and every workspace W, except those at

the leaves, has one or more inferior workspaces *Inferior(W)*. See Figure 3.2. The root workspace is the actual global database used to store archived products, libraries of components, fully validated designs, relationships among key features associated with the development process, processes for building the product, computer code, life cycle considerations, project organizations, etc. The workspace hierarchy supports the co-existence of different states of the same object. At any given time during the development process, the root workspace will contain the most recently released *public* collection of database objects (i.e., the omega release). "Super"-workspaces, that is, those closer to W_{root}, hold data which is more *correct*, *stable*, or *public*. The state of a design in a "sub"-workspace has a lesser degree of validation, is more tentative, or is less public.

The root workspace always exists. Other workspaces are *dynamically* created and destroyed. A sub-workspace may be created in order to separate unrelated projects or to create a work area with consistency requirements less stringent than those of the root workspace. In addition, a product developer or a group of product developers may also create sub-workspaces in order to encapsulate tentative or experimental updates, or narrow the focus to some subset of objects. Users can move into the context of the workspace hierarchy and examine the objects contained there.

Let W_i and W_j be workspaces. We define the $\leq$ relation between workspaces to be the reflexive and transitive closure of the *descendant* relation as follows.

$W_i \leq W_i$,

If $W_i \leq W_j$ then $W_i \leq Superior(W_j)$,

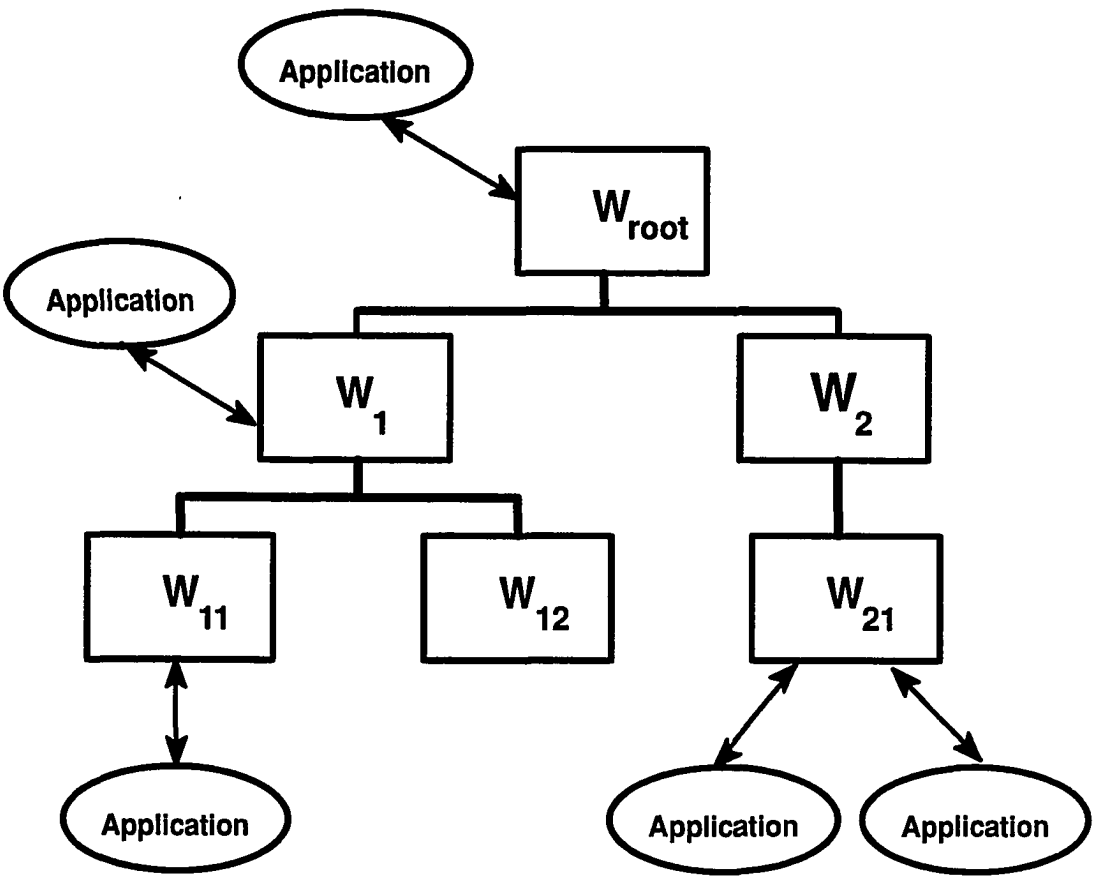

Figure 3.2: Workspace Hierarchy

If $W_i \leq W_j$ and $W_i \neq W_j$ then W_i is *sub-workspace*(W_j), and W_j is *super-workspace*(W_i).

The workspace hierarchy has invariants and semantics of commit and abort that are completely analogous with those presented in Section 3.3.3, given workspace $W \neq W_{root}$.

Along with the workspace hierarchy comes a generalized model for check-out and check-in. An object can be updated only in the workspace in which it is currently checked-out for update. (The rules of check-out and check-in will be described in detail later.)

In a database system which offers two levels of workspace: public and private (or experimental), the actions of check-out and check-in of an object strictly alternate [44]. The two-level workspace hierarchy does not allow for a natural representation of hierarchical tasks in which groups of users participate [20]. What is needed is a *dynamic hierarchy* of workspaces for users or user groups which permit a sub-workspace to be created *at any time*. In that sub-workspace, a subset of objects can be checked-out and *experimentally* updated without affecting the state of those objects in the superior workspace. When a set of updates is deemed acceptable, the objects can be checked-in and the changes are committed atomically to the superior workspace.

3.5.8 User mediated consistency

It has been recognized that one needs to create more flexible notions of consistency when dealing with product evolution. For example, Sutton [70] points out that when it comes to software development there are many reasons why one can-

not enforce consistency in the same way that one might in conventional database systems:

- it is difficult to discern and/or articulate all the constraints a-priori for a software system;
- automatic detection of all consistency violation (let alone automatic repair) is completely unrealistic;
- it is not always clear when one must check for consistency violations or where.

The functionality for dynamic constraint specification and collision records provide one answer to the above problems.

Constraint specifications

As previously stated, constraints among data must at some point in the development process be ascertained to be valid. One restriction which is intended to limit the propagation of potentially incorrect modifications to a design is the requirement that the validity of designated constraints of a design be ascertained before changes can be committed to a workspace. In conventional environments, this task is performed manually. The manual method is error prone; a user may forget to invoke tools to check consistency, or may be tempted to give intuitive (and maybe incorrect approval of the updates performed).

The proposed workspace hierarchy model offers *constraint specifications* in workspaces. A constraint specification is an attachment to a workspace that names a constraint in objects which must be known to be valid within an application's cache or in an inferior workspace before the application or inferior workspace can commit to that workspace. Constraint specifications are inherited by super-

workspaces.

Constraint specifications can be used to enforce some subset of constraints in certain workspaces in order to guarantee a known degree of consistency within that workspace. Different constraint specifications can also be assigned to different workspaces depending on the degree of correctness required. For example, a public workspace might have strict requirements, whereas an experimental workspace might have none. In essence, constraint specifications allow the exploitation of different correctness criteria for different groups and individuals.

A constraint specification does not determine how a constraint is to be validated, nor when. It is merely a restriction of committing changes to a workspace which is based upon the status of constraints. Other mechanisms are needed to control when to fire consistency checkers. The concept of cooperation motivates users intervention to amend constraint violation [28].

Collision records

Another crucial issue is that of *collision handling.* Even when users are benevolent and attempt to cooperate, there may be times when one user will make a change to an object that another user cannot understand, cannot adapt to, or considers an error, and therefore is unacceptable. We refer to this situation as a **collision.** Collisions may be identified when updates are applied to a shared workspace, or when an attempt is being made some time later to integrate a new version of a product.

When a collision occurs, a product developer or his/her application may wish to register its disapproval of the update in an effort to obtain corrective actions or

an explanation. This is done with a **collision record** that references the product developers involved in the collision, the update which caused the collision, and the application which performed the update. The product developers normally will try to resolve the collision between themselves. If they cannot, then resolution of the collision is the responsibility of a mediator. A *collision resolution* is a record that some action has been taken on behalf of the collision. A record of collisions and their resolutions should be kept for each workspace both in order to provide a history and to enable product developers and/or mediators to browse unresolved collisions [65].

In order to confine the out-spread of collisions, the cooperative product development environment should prohibit a workspace from committing to its superior workspace if it contains unresolved collisions. The conventional environment offers no support for collisions.

Collision records and constraint specifications provide our approach to implement user mediated correctness criteria.

3.5.9 Moniroting work status

Collision records are part of what is referred to here as *work status*. Other examples of work status include information pertaining to:

- the workspace hierarchy;
- the product developers participating in the different tasks;
- the active applications in different workspaces;
- the objects checked-out by different applications in different workspaces.

The ability to both access and track changes in work status is important for co-operating product developers since it provides a degree of *"awareness"* of what others are doing and hence it helps in monitoring the progress of the work and assists in coordinating the diverse efforts of the product developers participating in the process. As a result, supporting product developers with the ability to access and track changes in work status should be an integral part of a cooperative environment.

3.6 The Proposed Framework

The two preceding sections have described in what ways conventional databases and applications are inadequate for an environment that supports cooperation among product development groups. Enhancements to alleviate these deficiencies have also been proposed. Realizing these enhancements motivates our work. The remainder of this dissertation provides our framework for the enhanced capabilities.

The framework presents a software layer that resides between the data store and the applications which manipulate those data, thereby acting as an intermediary between the application and the data store. The framework is divided into two main components: the **Agent** and the **Cooperative Database Management System (or Co-DBMS)**. The agent consists of the application plus a set of software modules called the **Application Object Manager (or AOM)**. The Co-DBMS consists of an **object-oriented** data store with associated schema plus a set of software modules called the **Database Object Manager (or DOM)**. Operationally, the application within the agent invokes libraries of the agent which have been linked with the application - the AOM; the AOM interacts with the DOM in the Co-DBMS; and the DOM invokes functionality of the data store.

The model of interaction, described in Section 3.1, is modified as follows: product developers use agents to access and manipulate data stored by the Co-DBMS. See Figure 3.3. The modified model also assumes that product developers can (informally) communicate together to expose and reconcile differences in viewpoints. The assumption that applications are independent still holds. However, *awareness* of other applications is provided indirectly through messages received by an application, through its AOM, from the Co-DBMS (or more precisely, from the DOM within the Co-DBMS, as will be explained later) as a result of the actions of other agents.

3.6.1 Features of the framework

In order to provide services which are needed in cooperative environments, the framework exhibits the following features.

- **Object-oriented data store**

  All persistent data are stored in and are accessible from the Co-DBMS. In order both to control access to portions of data and to make manageable the amount of data which is transferred between agents and the Co-DBMS, data are divided into a large number of **interconnected objects**. Objects follow the object-oriented approach; each object has a type, an identity, an internal state, and a programmatic interface to access and change that state. The object model used by the framework is presented in Chapter 4. This object model constitutes the formal basis of our work.

- **Support for varying degrees of cooperation**

  Facilities provided by the framework accommodate sets of applications which

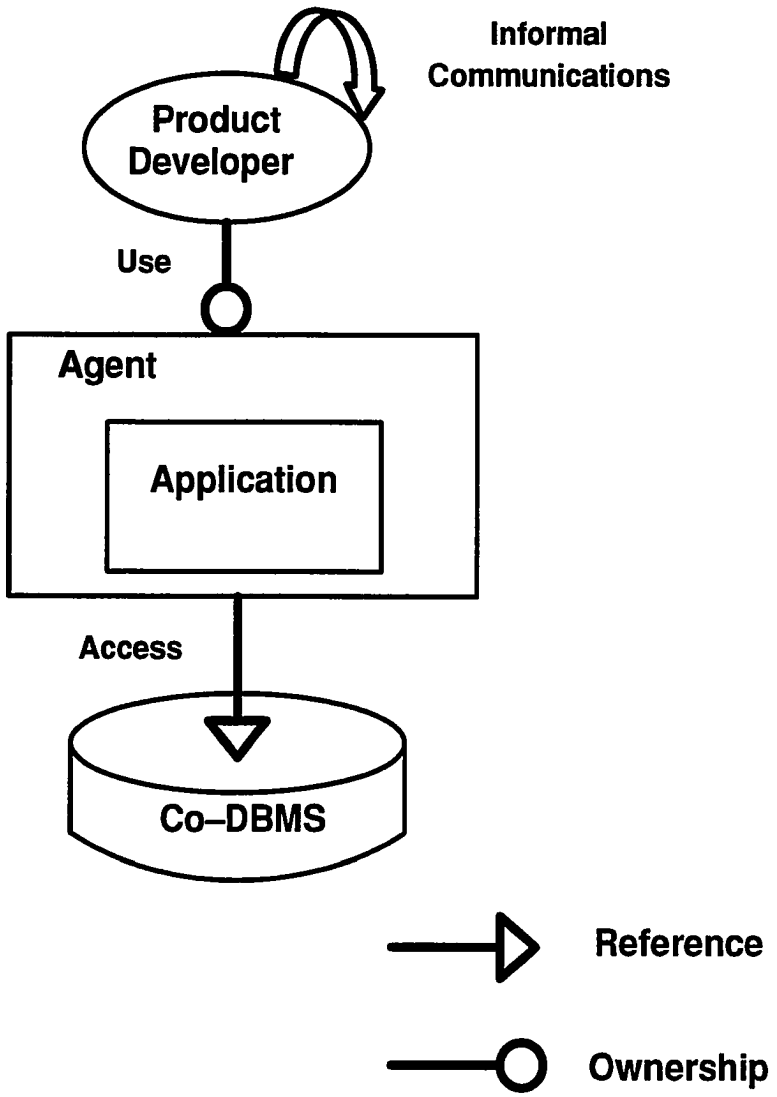

Figure 3.3: Modified High Level Interaction Model

are tightly coupled (such as a schematic editor and a simulator which are being used simultaneously by one product developer) as well as those which are loosely coupled (such as applications under the control of different product developers working on different aspects of the same artifact).

- **Use of notification**

  The Co-DBMS tracks updates to shared data. Unlike a conventional database in which the guarantee given to an application is that it has exclusive access to data, the Co-DBMS instead guarantees only that an agent will receive asynchronous notifications to maintain a view consistent with the database. Cooperating members of a group communicate informally; the DOM formalizes asynchronous communication between the Co-DBMS and the agents.

- **Extensibility and integration**

  The framework is *independent* of the semantics of particular applications within agents, so that new applications can be added to the environment without necessitating modification to the Co-DBMS or the agent software. In order to track changes to data in which the application is interested, each application informs its AOM of the set of updates in which it is interested. When an event occurs which matches an interest, the application which registered the interest is sent a **local notification.** If another agent updates an object in which an application is interested, the AOM in the agent of that application will be sent an **external notification** by the Co-DBMS. The AOM offers a *uniform* programmatic interface and associated protocols, with which applications can create, destroy, commit, and abort workspaces, check-out and check-in objects, and access and update data.

- **Dynamic workspace hierarchy**

  The Co-DBMS offers a hierarchy of workspaces and associated check-out and

check-in protocols with which to encapsulate tentative changes to objects. Although many users may have permission to update the same object, that object can be checked-out in at most one workspace at any time. Thus if two users wish to update some object at the same time they must do so within the same workspace.

- **Enforcing constraint specifications**

  The Co-DBMS enforces consistency specifications attached to workspaces.

- **Collision mechanism**

  The framework offers applications a mechanism to register collisions and their resolutions, and prevents updates in an inferior workspace from being committed to its superior workspace if there are unresolved collisions in the inferior workspace. The framework does not enforce a particular policy of collision resolution but rather provide a vehicle for instituting policy by allowing applications both to decide which changes constitute collisions and to determine what is done in the event of a collision.

- **Work status monitoring**

  The framework gives applications access to the work status. Work status is maintained by the Co-DBMS and made available to all agents.

- **Automatic agent cache consistency**

  Agents cache objects which they are accessing in an **extended object cache** (explained in Chapter 5), which is the agent's local object workspace. A cache may grow stale, however, when another agent updates those objects. The AOM in the agent processes update notifications from the DOM in the Co-DBMS and ensures that the cache stays consistent with the Co-DBMS in the face of updates by other applications.

- **Automatic invalidation of constraints**

  It is unreasonable to assume that applications will understand the impact of updates they make on all constraints associated with the product and its development process. The set of constraints may grow over time, for example, as the data model evolves. For this reason the AOM is responsible for invalidating constraints whose validity may have been disturbed by updates.

- **Efficiency**

  Modules of the AOM are directly linked with the application in an agent. When these modules are invoked by an application, the CPU of the workstation running the agent is used. The DOM manipulates and controls access to data in workspaces, but the manipulation of data are performed by the individual applications, each with its own set of special-purpose data structures which enable it to perform its task efficiently.

3.6.2 Architecture and operation

Remaining chapters of this dissertation present in detail the architecture and operation of the DOM and the AOM, explain what capabilities they add to the DBMS and applications, respectively, and show how these capabilities provide what is needed in cooperative product development environments.

Chapter 4

The Object Model

The notion of an object from object-oriented programming languages (OOPLs) and object-oriented databases (OODBs) provides a way to describe all of the complex data that are usually required in cooperative product development environments such as CAD/CAM and CASE [10, 39, 67, 50, 44, 55, 17]. Applications in these environments manipulate data that are often complex and intricately connected by numerous consistency constraints. For example, in software development, the notion of an object is sufficiently powerful to describe things as diverse as program modules, test cases, compilation, specifications, and documentation, so it provides a natural uniform way of describing the artifacts and processes of software engineering.

In this chapter, we propose a **data model** for cooperative product development environments, intended for applications such as VLSI circuit design, mechanical parts design, and software development. The context for our data model is **object-oriented** in which data are broken into a collection of **interrelated objects.**

This chapter describes the characteristics of the object-oriented data model,

presents and explains the schema language which is used to define a specific data model, and discusses the ways in which objects may be related to one another. It then enumerates what operations can be performed on objects, and offers a detailed example of a simple schema from the software development domain.

4.1 The Object-Oriented Approach

There are three basic concepts to object-oriented modeling: **objects, types**, and **messages**. Briefly, objects are the building blocks that combine data and processes to perform a specialized role in the system; a type is a template for similar objects; while, messages represent the interface that allows objects to interact without having to understand or interfere with others' internal processes [73]. In this section, we describe each of these concepts with emphases on the characteristics that serve our model.

4.1.1 Objects

Although there is no common definition of object, we present here a working definition for the purposes of our work. An **object** is an entity that encapsulates **state** and **behavior** into a self-contained package. Every object has the capability of storing data, which define the state of the object. The behavior of an object defines the ways in which the object's state can be affected. Objects are created and destroyed dynamically. The lifetime of an object is independent of the lifetimes of other objects, (except in some case of the object being owned by another object; this case will be explained later in the chapter). Objects have three key properties: identity, state, and operations (or methods). Each of these is described below.

- **Object identity**: every object is an abstraction that has an identity that is independent of the values of any of its properties or relationships to other

objects. This identity is captured by a unique, immutable **object identifier (OID)**. Basically, an OID is an arbitrary numerical value, that is automatically assigned and maintained by the system, and the system ensures its uniqueness. The OID is used as a handle with which a client of the database system (such as an agent) can reference and access the corresponding object.

- **State**: objects of the same type are specified and distinguished by their **state**, which may (or may not, for some kinds of objects) change over time. The state of an object is captured and maintained in **named slots (or variables)**. Each object has an array of slots to store state data. A slot's value can be specified to be either single or multi-valued.

  Each slot has a **name, type** and **value**. Slot names are unique within an object type. The slot type designates the type of the value that can be assigned to that slot in instances of the object type in which this slot is declared.

  A slot of one object can be referenced by another object (described below). Each slot, however, is owned by exactly one object and is not shared. Moreover, updates to a slot must be done through the object which contains it.

- **Operations**: an operation (or method) is a mapping from some input objects to output objects. The mapping is performed in response to messages sent by other objects. Operations are applied to create or destroy objects, to access their attributes, to compute results, to test constraints, or to trace relationships to other objects. An operation is executed only when the correct type of message is received from the right source object. Only the object's operations can access its state.

Operations are embedded within objects rather than operating as free-standing entities. Each operation that can be applied to an object has a **name**, an optional set of **parameters**, and a **body** (implementation). Operation names (and parameters) are known externally. The body is known only to the containing object. The body is a *procedure*, written in some programming language, that is executed (by the containing object) when the corresponding operation is triggered. This procedure, which can access or change the state of the object, performs the mapping from inputs to outputs, and may have messages sent to other objects from within. The object processing an operation, first completes that operation before receiving any more messages.

4.1.2 Types

Objects are associated with types (or classes). A type is an abstraction that allows the user to encapsulate similar objects. An object type is simply a template for those objects exhibiting similar characteristics, and it defines the aspects of objects that are the same for all the actual realizations (objects) of that type. Consequently, the object type determines what slots the objects of that type have, and the operations to be applied to those objects. Objects of a given type are called **instances** of that type. Instances of a type are related to that type through the "**is-a**" relationship. For example, all objects whose state and behavior correspond to the common notion of rectangle, are instances of type "rectangle." Similarly, one can have types layout, queue, etc.

Many different types can be defined to serve different purposes. The various types, however, are not defined in isolation. Rather, they are defined as special cases of each other, forming what is known as a **type hierarchy**. For example, the

collection of products a company offers could all be defined as specialized versions of more general products, all of which could be considered special cases of the more general type *product*. Formally, these special cases are known as **sub-types**. The types of which they are special cases, in turn, are known as their **super-types** [73].

The advantage of defining types in a hierarchy is that, through a mechanism called **inheritance** [73], sub-types share all the characteristics of their super-types. For example, a "NAND gate" would inherit all the operations and variables of its super-type "Gate".

4.1.3 Messages

An object-oriented computation proceeds by **messages** sent from one object to another. By convention, the object sending the message is called the **sender** and the object receiving the message is called the **receiver**.

Structurally, a message consists of three parts: the identity of the receiver, the operation name the receiver is being asked to carry out, and a list of (optional) parameters that the receiver may need to perform the requested operation. Two special object types are generally recognized: ANY – to indicate any from the universe of object types; and SELF – to indicate the object that issued the message itself.

Using messages to carry out interactions between objects confers the same benefits as in real world – namely, it protects the internals of objects from outside intrusion, and it protects all the other objects from having to contain information about the structure of any one object. Another benefit of using messages is **polymorphism** [73] – because objects are defined independently of one another, the

same name can be used for different operations in different objects. To illustrate polymorphism, consider, for example, the message "add": sent to a purchase order, it might mean add a new line item; meanwhile, if it is sent to an account object, it could be an instruction to increase the current balance.

4.2 The Proposed Object Model

The object model we propose adopts the object-oriented approach (desrcibed in Section 4.1). In addition, objects in our model are related through various relationships. In this section, we describe the object schema, the different relationships among objects, and the operations on objects.

4.2.1 The object schema

The particular data model offered by the database system will depend on the application domain chosen and upon design decisions made by the person(s) who define(s) the data model. The data model in use, that is, the object types, the structure and types of their slots, and the operations applied is described by the **schema**. The syntax of a schema is presented below in Backus-Naur Form (BNF).

- The schema consists of a number of declarations of object types.

  <Schema> ::= <schemaDecl>

  <schemaDecl> ::= <objectTypeDecl> | <objectTypeDecl> <schemaDecl>

- A declaration of an object type specifies the name of the object type followed by declarations of the slots that will capture the state of objects of that type.

<objectTypeDecl> ::= <objectTypeName> { <Slots> }

<Slots> ::= <slotDecl> | <slotDecl> ; <Slots>

- Object types are named by identifiers, or character strings chosen by the person who defines the data model. Object type names are unique within the database.

  <objectTypeName> ::= identifier

- The declaration of a slot consists of a slot name, followed by the type of the value that can be assigned to that slot in instances of the object type in which this slot is declared.

  <slotDecl> ::= <slotName> : <slotType>

- Slots are named by identifiers. A slot name is unique within an object type.

  <slotName> ::= identifier

- A slot's value either can be assigned any of a specified type, or can be the result of a computation applied to the values of other slots (a **derived** slot).

  <slotType> ::= <typeDecl> | **derived** <derivationDecl>

- A slot's value can be specified to be either of a **basic type** (e.g., logical, integer, real, string, etc.), an **object type**, in which case the value is a sub-object of the object which owns the slot, a **set** of values of a specified type, or a **reference** to another object of a specified type.

  <typeDecl> ::= <basicType> | <objectTypeName> | **set** <typeDecl>
  | **reference** <objectTypeName>

  There are important differences between a slot's value being a sub-object, and its value being a reference to another object: in the former case, the lifetime of the sub-object is tied to that of the containing object in that the sub-object is created or destroyed when the containing object is created or destroyed, respectively; in the case of a reference to an object, the lifetimes of the referencing and referenced objects are unrelated, in this case, referential integrity is enforced, however, which means that an object cannot be destroyed if another object references it.

- We present here four basic types: **logical** (true or false), **integer** and **real** (numeric), and **string** (array of characters). It is to be noted, however, that these types are merely examples. Other types might be added and are absent only for the sake of simplicity.

  <basicType> ::= **logical** | **integer** | **real** | **string**

- The computation of a derived slot value either can be the responsibility of the agents (**external**) or can be automatically carried out by the system

(**direct**). In the former case, the computation applied to the values of other slots to obtain the derived slot value is usually arbitrary and may be potentially complex. In the later case, on the other hand, the computation is fairly simple such as having the value of the derived slot to be equal to another object or a slot of another object.

<derivationDecl> ::= **external** <externalSpec> | **direct** <directSpec>

- To explicitly state that the value of a slot, representing some aspect of the object, depends on the values of other slots in a way that might require some arbitrary computation, we introduce the concept of **derived external** slots. The system does not have the capability to automatically keep these derived values current. Such derivations are the responsibility of the users. The schema, however, indicates in the <externalSpec> the slot names [<slotNames>] upon which the **derived external** slot depends.

  <externalSpec> ::= <typeDecl> [<slotNames>]

  <slotNames> ::= <slotName> | <slotName> , <slotNames>

- The value of a derived slot can be directly computed from sub-objects or referenced objects. The <derivationformula> specifies how that value is to be obtained. For simplicity, we only consider cases where the value of the derived slot is a copy of sub-objects or referenced objects; the **derived direct** slot may assume the value of the slot of a sub-object, which, in case of a set-valued slot, would result in a set of values, or it can be the result of following a reference to another object. (Derived slots are presented in detail

in Section 4.2.3.)

<directSpec> ::= <derivationformula>

<derivationformula> ::= <slotName> . <slotName> | <slotName>↑

Where $X.S$ denotes slot S is a sub-object of slot X and $X\uparrow$ denotes a reference to slot X.

4.2.2 Relationships among objects

Although objects are independent entities with their own separate identities, objects can be related to each other via relationships. Relationships are one of the most fundamental parts of any data model. From one point of view, they are what distinguish databases from file systems [50]. The participation of objects in a relationship is defined by mappings, where a relationship has a mapping for each one of the object types it relates. Relationships among objects are expressed through object slots.

In this subsection, we will elaborate two ways in which objects can be related to one another: **composition** and **reference**. We view these as being crucial to cooperative product development environments.

Composition

It is possible for an object X to be nested within another object Y, as defined by the data model. In this case the nested object X is said to be a sub-object of Y that is contained in Y, and Y is said to be the owner (or container) object. A sub-object is related to its owner object by the **is-a-part-of** relationship. An

object can be a sub-object in either of two ways: it can be the value of the slot of another object, or it can be a member of a set-valued slot of another object. Set-valued slots are useful when the number of constituent objects cannot be determined in advance [48, 73]; an example is a design which contains some number of components.

An object which is composed of other objects can itself be nested in an object. Thus recursive nesting of objects can give objects a hierarchical structure. Object composition is acyclic. A given sub-object can have *at most one* owner, and its owner (if it has one) is fixed for the lifetime of that object. Having the lifetime of a sub-object to be tied to its owner, implies that when an object is created, sub-objects are also created (except in the case of a set-valued slot, that would initially be empty); when an object is destroyed, so are its sub-objects (and in the case of a set-valued slot, all sub-objects in that set).

Of particular interest is the object that is not contained in any other object. This we refer to as a **base object**. A base object has a lifetime which is independent of any other object. It follows, from the above presentation, that if object X is not a base object, it is contained in a single other object Y, where $X \subseteq Y$. We then say that X is a *sub-object* of Y. Accordingly,

> *the database can be viewed as a collection of base objects that can be linked together through references.*

References are explained next.

References

When an object is contained in another object, it is accessible only through its owner. Also, composition provides a way to tying an object's lifetime to that of

the container object. There is another way to make an object accessible to another object: through a **reference**. A reference is a handle to an object by which the object can be accessed by other objects. References to an object are stored in the slots of another object. If an object X or a sub-object of X references object Y or a sub-object of Y, then X is said to reference Y, denoted $reference(X, Y)$. Abstractly, we can view an object as having two types of slots: descriptive slots - describing characteristics of the object components, and reference slots – linking the object to other objects. In addition, each slot, whether descriptive or reference, may be set-valued.

References are useful because they permit sharing of information. In a CAD database, for example, components within one or more designs may reference the same design because instances of that design appear multiple times within the parent design(s). The referenced object represents a common substructure of all objects which reference it. Another benefit of references is that the referenced objects can change in size and composition without affecting the referencing objects.

When an object is destroyed, all references it has to other objects are also destroyed. **"Referential integrity"** is enforced, however, an object cannot be destroyed if there are references to it. Unlike composition, an object is free to have any number of references to it, also object references may be cyclic.

4.2.3 Derived slots

Sometimes the value of a slot S may be related to other slots – called *source* slots – in that if any of those slots change, then the value in S may also need to change in order to stay current with its source slots. Such a slot is called a **derived slot.** Derived slots are used as a means of explicitly specifiying the the semantics of the

relationships among different objects. A **derivation specification** is designated for each derived slot. In our data model we distinguish two types of derived slots: **derived direct** and **derived external** slots. The rest of this section explains the semantics of both types.

Direct derivation

In general, the value of a slot can be specified to be the same as that of another object, or the sub-object(s) of an object, or the result of following a reference to another object, or some combination of the preceding. The derivation specification of a derived direct slot consists of a set of source objects and a derivation function determining how the value is to be obtained from this set. The value of derived direct slots, unlike derived external slots (see below), always stay current with respect to their source slots and this currency is maintained by the system.

External derivation

The above stated derivation functions are simple and can be quickly recomputed when one of the source slots change. There are some cases, however, where the computations may be arbitrary and potentially complex [56]. Consider, for example, the case where the derivation function has to run design consistency checking and/or perform some analysis. Our solution in this case, is to have the derivation specification include only the source slots and leave the computation to be carried out by the tools most preferred by the users. We identify this special case of derived slots as **derived external** slots. Which tools to use might as well be mentioned in the specification, however, this may be less accommodating to users' preferences to using particular tools.

In addition to the above usage of derived external slots, they can also be utilized

in situations where it is difficult to determine automatically when a given change to an object *necessitates* a change to a potential dependent. Therefore, derived external slots provide us with the ability to define a *weaker* notion of dependency; capturing only that the dependent object *might* require change, but defining neither the circumstances under which a change is absolutely required nor the nature of the change.

4.2.4 Operations on objects

Base objects can be created and destroyed. Updates to the state of an object are accomplished by making updates to its slots. Updates are performed by applications on their own cached copies of objects, as will be described in Chapter 6. In this section, we present those operations that create and destroy base objects, and describe what updates on slots of objects are permissible for each type of slots.

Operations on base objects

Create

Base objects are created (instantiated) by means of the *Create* operation. The newly created object is given a unique OID; its slots assume default values. Originally, the identity of the created object is only known to its creator. It may, however, be passed to other objects as part of an attribute list in a message.

Destroy

A base object removal is accomplished via the *Destroy* operation. When an object is destroyed, all of its bound sub-objects are destroyed. Referential integrity requires that an object can be destroyed only if it is not referenced by another object. In the case of a group of objects participating in a circular reference, the circularity must be broken by changing one or more of the references before any

object in the group can be removed. Before physical removal, a clean up takes place.

Restore

A base object can be restored before clean up. This means that the effect of previously destroying the object has been undone. The only restriction is that references to non-existing objects are nullified. Restoring an object is similar to creating a new one, except that the identity and state of the restored object are the same as those before the object was destroyed.

Operations on object slots

Basic slot

The only operation available on a basic slot is that of assignment of a value v to slot S:

$X.S := v$, where v is of the appropriate basic type.

Sub-object

If S is a sub-object of X, then the updates possible on S are those possible on any slot of S, as described in this section.

Set of sub-objects

If S is set-valued, then any of the following updates is possible:

create new member in $X.S$

destroy member $Y \in X.S$

restore member $Y \in X.S$

update $Y \in X.S$, as described in this section.

Reference

If S is a reference to another object, then either that reference can be destroyed or replaced with a different reference:

$X.S := nil$, which nullifies any existing reference, or

$X.S := \uparrow Y$, which assigns a reference to object Y to slot S.

Derived direct slot

No updates are permissible on a derived direct slot, since its value is automatically assigned whenever any of its source slots changes.

Derived external slot

A derived external slot S of object X is assigned a value as follows:

$X.S := v$, where v is of the appropriate basic type.

4.2.5 Dependencies among objects

Let X, Y, and Z be base objects. We define the *dependson* relation between base objects as the transitive closure of *references*:

Let $dependson(O1, O2)$ denote object $O1$ depends on object $O2$. Then we have

$dependson(X, X)$,

$dependson(X, Y)$ and $reference(Y, Z)$ implies $dependson(X, Z)$

Given the definition of *dependson*, we now define "sources of X" $sources(X)$ and "dependents of X" $dependents(X)$ as follows:

$sources(X) \equiv \{all\ Y : dependson(X, Y)\}$,

$dependents(X) \equiv \{all\ Z : dependson(Z, X)\}$.

Finally, we define "object group of X" $objectgroup(X)$ as:

$objectgroup(X) \equiv \{\ all\ Y : Y \in (sources(X) \cup dependents(X))\}$.

The *dependson* relation, and subsequently references, are of particular importance to defining conflicts in a principled manner. The *dependson* relation is used in our model to *explicitly* depict the fact that if an object is changed, then its dependents might need to be altered as well. The state of object Z can be affected by the change in the state of object X **only if** $dependson(Z, X)$, or alternatively $Z \in dependents(X)$. The use of the *dependson* relation in concurrency control will be explained in detail in Chapters 5 and 6.

4.3 Example

This section gives a simplistic example of a schema and associated objects. The example is chosen from the software development domain. Other schemata will be used for other domains.

Schema

In this schema, a program is built from subroutines and libraries. Subroutines are either contained locally or are external to the program. The executable code of a program is the result of linking compiled subroutines with libraries. Subroutines are compiled from their source code, and libraries contain compiled object code. This example makes use of three object types, five derived direct slots, and two derived external slots.

Program

{

suboroutineRefs: **set ref** Subroutine

libraryRefs: **set ref** Library

subroutineLocals: **set** Subroutine

subroutineExternals: **derived direct** suboroutineRefs↑

subLocalsObjCode: **derived direct** subroutineLocals.objCode

subExternalsObjCode: **derived direct** subroutineExternals.objCode

libraries: **derived direct** libraryRefs↑

libObjectCode: **derived direct** libraries.objCode

executable: **derived external bytes**

[subLocalsObjCode, subExternalsObjCode, libObjectCode]

}

A program's executable code is computed from compiled subroutines and libraries. If the object code associated with a subroutine or library changes, the executable is out-dated and must be recomputed.

Subroutine

{

srcCode: **ASCII**

objCode: **derived external bytes** [srcCode]

}

A subroutine has two parts: source code, and object code computed from the source code. If the source code of a subroutine changes, its object code is marked out-dated and must be recomputed. The "ASCII" and "bytes" designations for source and object code, respectively, merely indicate basic types of data that have no semantic meaning to the database; "ASCII" would probably contain ASCII text, and "bytes" would probably contain machine instructions.

Library

{

objCode: **bytes**

}

A library consists of pre-compiled object code.

Chapter 5

The Cooperative Database Management System

The Cooperative Database Management System (Co-DBMS) is proposed as part of our framework to support cooperative product development. It follows the paradigm of a server, whose function is to await and service requests from clients, in this case agents. The Co-DBMS is unlike a server, however, in that servicing a request from one agent may cause asynchronous notifications to be sent to other agents. This chapter presents the architecture of the Co-DBMS, describes what functionality it adds to an object-oriented data store in order to overcome the weaknesses discussed in Chapter 3, presents the programmatic interface between agents and the Co-DBMS, and summarizes the rules maintained by the Co-DBMS.

5.1 Architecture of the Cooperative Database Management System

The Co-DBMS consists of an object store plus seven modules: timer, agent information manager, workspace manager, object access manager, collision records

manager, update monitor, and work status monitor. See Figure 5.1. The object store provides persistent storage of the schema, extended objects (explained in Section 5.2.7), and access permissions to those objects. The seven modules constitute the proposed Database Object Manager (DOM). These modules interact with each other and together with the object store offer a collection of services to agents.

Product developers, in the environment, will run applications locally on their own workstations. Applications are run asynchronously with respect to one another. The applications will communicate with the Co-DBMS through the use of inter-process communication (IPC) [71]. The latency of IPC is high compared to communication within a workstation. So the choice of what functionality to assign to the Co-DBMS has been motivated by the need to reduce the frequency of interaction of applications with the Co-DBMS. In order to accomplish this, the programmatic interfaces presented in this chapter specify a granule of operation at the level of *base objects*, rather than at the level of *slots*.

The Co-DBMS Timer

The Co-DBMS maintains an integer-valued timer. The timer is incremented whenever the DOM processes a request from any agent. If a request contains sub-requests, such as when an agent commits a collection of updates to the Co-DBMS, the timer is incremented once per sub-request. If the DOM receives requests from multiple agents at the same time, the requests are queued to be processed following desired queuing policies (out of the scope of this work).

The time-stamp of some requests, such as an agent committing a batch of updates, is remembered by the Co-DBMS for later use. Other time-stamps such

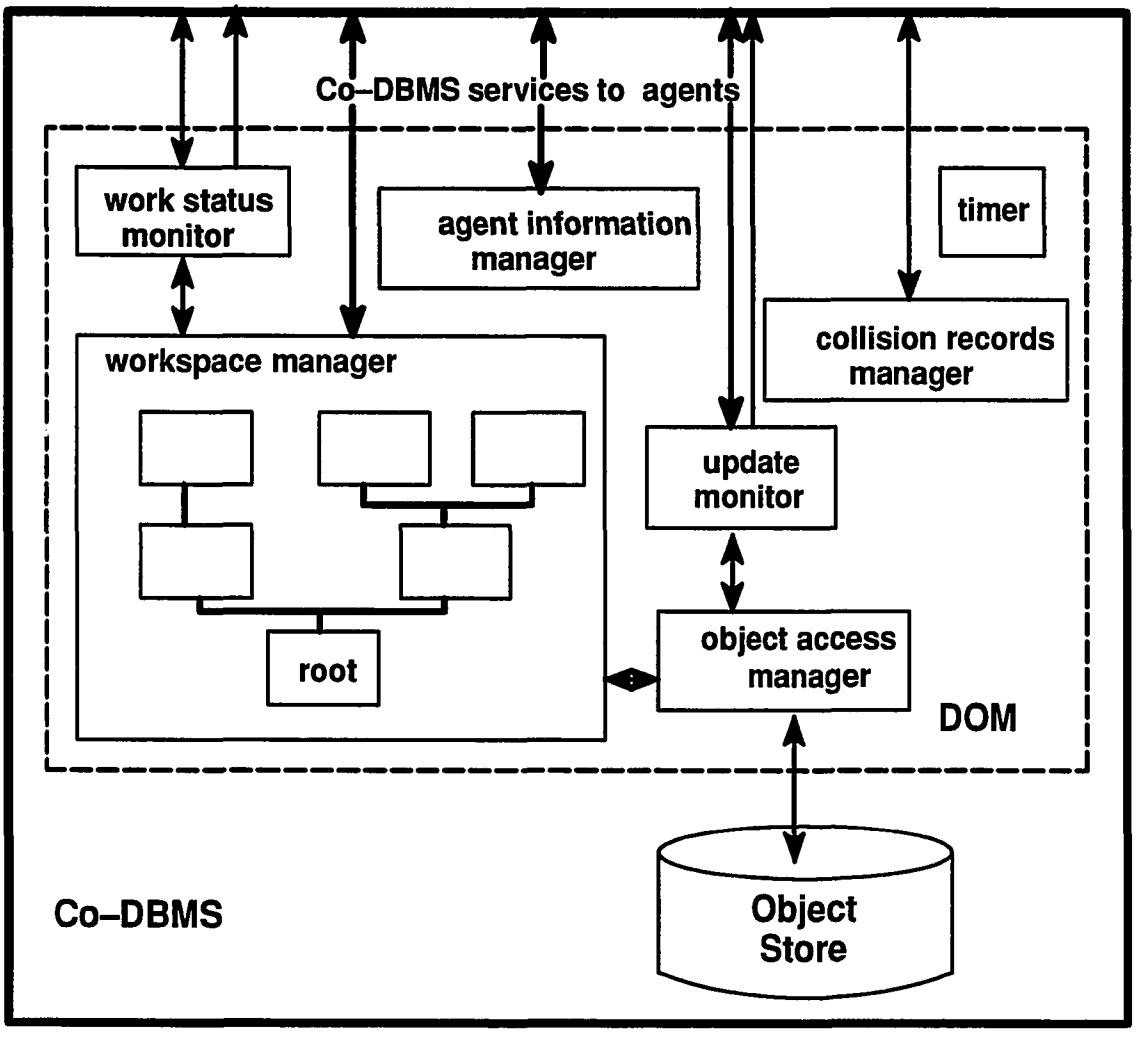

Figure 5.1: Architecture of the Co–DBMS

as the time at which the slot of an object changed value, are stored in the object store. Because the timer is incremented after each request, time-stamps are unique. If the Co-DBMS is distributed, then methods for event ordering can be used to ensure uniqueness of time-stamps [47].

5.2 Functionality of the Database Object Manager

This section describes each service offered by the DOM to the agents, presents the interface which an agent uses to access the service, and explains how modules within the DOM operate in order to provide that service.

5.2.1 Connecting agents to the DOM

Some of the information maintained by the DOM, such as which objects have been checked out, is associated with a particular instance of an agent. Thus each instance of an agent has its own identity. That identity is established when an agent is first connected to the DOM, and is removed when an agent is disconnected.

ConnectAgent(UserName, AgentName)

return AgentID

When an agent starts execution, it must be registered with the DOM. The DOM records what agent is running and what user is operating the agent, and returns an agent identifier (AgentID) which uniquely identifies that instance of the agent. The name of the agent and the name of the user can be accessed as part of the work status, as described in Section 5.2.6. The AgentID returned by the DOM is used in subsequent requests to the DOM to identify the agent making the request.

DisconnectAgent(AgentID)

When an agent terminates, it must be disconnected. The DOM invalidates the agent ID and removes the name of the agent and the user from the agent register (the list of currently executing agents in the work status). If an agent has a workspace selected (described in Section 5.2.3), this workspace must be unselected before the agent can be disconnected.

5.2.2 Creating and destroying workspaces

As noted in Section 4.5.7, a dynamic workspace hierarchy is useful in cooperative product development. Our system offers agents the ability to create and destroy workspaces, and to determine what workspaces exist.

CreateWorkspace(AgentID, SuperiorWorkspaceID, Description [,*set* InferiorWorkspaceID])
return WorkspaceID

This operation creates an inferior workspace of a specified superior workspace. The name and description of the new workspace are given by the creator. A set of workspaces which are inferior to the specified superior workspace can optionally be supplied; doing so will make them inferiors to the new workspace.

Initially, the collection of objects viewed from the new workspace is identical to the objects viewed from the superior workspace, and the set of constraint specifications is the union of those of its inferior workspaces (or the empty set if no initial inferior workspaces were specified).

DestroyWorkspace(AgentID, WorkspaceID)

This operation destroys a specified workspace. A workspace W can be destroyed only if the following three conditions are met:

1. $W \neq W_{root}$, since the root workspace always exists;
2. no currently executing agent has W selected (defined in Section 5.2.3);
3. there are no uncommitted changes in W.

If there are uncommitted changes in W, W must first be committed or aborted before being destroyed. When a workspace is destroyed, its inferiors may either be destroyed or become the inferiors of its superior workspace.

ListWorkspaceInferiors(AgentID, WorkspaceID)
return set WorkspaceIDs

Given the workspace identifier, the list of inferior workspaces is returned. This command assists the agents in traversing the workspace hierarchy.

5.2.3 Workspace selection

All accesses and manipulations of objects must be performed within a particular workspace. Multiple agents can simultaneously operate in the same workspace. The choice of a workspace depends on the degree of cooperation and interaction desired with other product developers and their agents. When two agents share a workspace, they can work together more closely and share updates to objects in a

less restrictive manner. The choice of a workspace also depends on how stable a view of objects is needed by the agent.

An agent must inform the DOM in which workspace it needs to operate; this is called **workspace selection**. An agent that has selected a workspace W performs operations associated with W until it *explicitly* unselects W.

An agent may have responsibilities in more than one workspace. This enables agents to have different contexts simultaneously, and to use workspaces alternatively. The agent can allocate time and move between these workspaces as priorities and deadlines dictate. However, as an active agent A in a workspace W_i, A cannot select another workspace W_j even if it has responsibilities in W_j. Moving from one workspace to the other means unselecting the first before selecting the second.

SelectWorkspace(AgentID, WorkspaceID)

This operation allows an agent to select the context of a particular workspace in which to access the database. With no workspace selected, an agent cannot check-out objects. An agent can have *at most one* workspace selected at any given time.

UnselectWorkspace(AgentID)

UnselectWorkspace informs the DOM that an agent has finished working in a workspace. Before an agent can unselect a workspace, it must check-in any objects that it has checked-out.

5.2.4 Constraint specification

As mentioned earlier, constraint specifications, which are attached to a workspace, specify that a certain subset of constraints must be met both before and after any set of changes is applied to objects in this workspace. This facility can be used to ensure that artifacts meet certain standards before they are admitted to superior workspaces that are more publically accessible. Constraint specifications provide assurance to any product developer working in a particular workspace that the objects in that workspace conform to a certain level of validation.

Constraint specifications are enforced by rejecting a set of updates by an agent to the workspace if one or more of those constraints is invalid. In addition, an inferior workspace is also prevented from committing to that workspace if in the inferior workspace one or more of those constraints is invalid.

Workspaces inherit constraint specifications from inferior workspaces. Thus, the set of constraint specifications for a superior workspace is a superset of those for its inferiors, which means that the degree of correctness required of a superior workspace is at least as stringent as that required of its inferior workspaces. The root workspace, since it holds objects which have achieved the highest level of validation, has a large number of constraint specifications. Normally, agents will operate in workspaces which have a few constraint specifications, in order to make interactive updates during which no particular degree of consistency of the artifact is expected to have been achieved. Constraint specifications can be added, removed, or queried as follows.

AddConstraintSpecification(AgentID, WorkspaceID, ObjectType,
ConstraintSpecification)

return ConstraintID

This operation adds a constraint specification to a specified workspace. The constraint is specified as a slot of a particular object type; the slot must be of type logical. A constraint specification cannot be added to a workspace unless the constraint is met by all objects of that type in that workspace and in all super-workspaces.

RemoveConstraintSpecification(AgentID, WorkspaceID, ObjectType,
ConstraintID)

RemoveConstraintSpecification removes a constraint specification from a specified workspace and all sub-workspaces. Subsequent updates to those workspaces are accepted by the Co-DBMS even if the value of the slot is **false**.

ListConstraintSpecification(AgentID, WorkspaceID)
return set Constraint

This operation returns the set of constraint specifications attached to a particular workspace.

5.2.5 Collision recording

Collision recording is needed in cooperative product development environments where concurrent work is inherent. A collision record is attached to each workspace to help product developers resolve their collisions. The operations used to service collisions are as follows.

RecordCollision(AgentID, OffendingAgentID, Complaint)

return CollisionID

An agent invokes RecordCollision to register a collision about an update made by another agent. That agent identifies the offending agent and supplies an explanation of how the update constitutes a collision. The DOM does not understand the semantics of collisions, and can make no attempt to remedy the collision; it only provides the *mechanism* with which collisions can be recorded (higher level mechanisms are needed for implementing particular policies). Collisions are recorded in the workspace which the complaining agent has selected. If a workspace has unresolved collisions, it is not allowed to commit.

ResolveCollision(AgentID, CollisionID, Resolution)

After some action has been taken to remedy a collision, it can be marked as having been resolved; an explanation of how the collision is resolved is supplied using the ResolveCollision operation. After all collisions are resolved, updates in a workspace can be committed to the superior workspace. The DOM guarantees that recorded collisions are not lost, but provides no assurance that a responder handles the resolution of a conflict correctly.

ListCollision (AgentID, WorkspaceID)

return set Collision

This operation returns the set of collisions, each with its associated resolution (if exists), that took place in a specific workspace.

5.2.6 Work status monitoring

We consider maintaining work status to be of great relevance in cooperative product development. Knowledge of work status can aid agents in the task of planning and coordinating their activities. A mechanism is provided with which agents can access and stay aware of changes to the work status. Higher level mechanisms can use this mechanism in order to implement policies of work methodologies or shared access.

The work status made available to agents is mainly the internal state of the system that can be altered by invocations of the operations that are presented throughout this chapter, specifically:

- which agents are currently running;
- what is the hierarchy of workspaces;
- which agents have selected what workspaces;
- the constraint specifications attached to a workspace;
- which agents are involved in what roles in what workspaces;
- what workspaces have uncommitted updates;
- what collisions and collision resolutions have been recorded in a workspace;
- what objects have been checked-out by what agents.

GetWorkStatus(AgentID, WhichStatusReport)
return WorkStatusRequested

Agents can obtain any of the above work status by invoking GetWorkStatus and specifying what work status is needed.

TrackWorkStatus(AgentID, WhichStatusReport, WhichChange)
return StatusID

At times it may be useful for an agent not only to obtain work status, but to track changes to it as well. For example, an agent might like to know when another agent has checked-out the same object. An agent specifies in what changes to which work status it is interested by calling TrackWorkStatus. The agent will receive asynchronous notifications of changes to the specified work status until the agent cancels its interest in tracking that work status.

StoptrackWorkStatus(AgentID, StatusID)

Using StoptrackWorkStatus, the agent indicates that it no longer wishes to receive notifications of those changes to a specific work status.

5.2.7 Committing and aborting workspaces

When objects in a workspace achieve some desired state, it is useful to make them more public or to move them to a workspace used for integration with the efforts of other agents. This is achieved by committing the workspace to its superior workspace. Immediately after the commit, all objects in the committed workspace and its superior have the same state.

To implement the commit operation, the DOM maintains for each workspace W in the workspace hierarchy ancillary information, in the form of attributes

called **control attributes**, which it uses to compute the update delta between *Superior*(W) and W. When an object is augmented with this control information it is termed an **extended object.**

Extended objects

The object store provides persistence for objects, that is it stores the values of their slots. So that the DOM can remember what updates have been applied to each workspace, the object store holds additional information for each workspace about each object that has been altered in the workspace. The result is an **extended object.** The information is held at the level of each object unit (i.e., the base object level, the slot level, and the value level). The control attributes of base objects and of each slot and value of base objects are described in Table 5.1.

When a workspace W is committed, the DOM scans the objects in the workspace and uses the control information to determine what objects and set members were created or destroyed and what slots changed in order to generate a collection of updates that represent the update delta for the workspace. That update delta is then applied to *Superior*(W) and the extended objects in W are discarded; they are no longer needed because the objects in *Superior*(W) now have the same state as they did in W.

When a workspace W is aborted, no update delta is created. Instead, extended objects in W and all of its sub-workspaces, are simply discarded. A workspace is aborted only if the updates that have been applied to objects in the workspace are to be undone.

object unit	control attribute	value	meaning
base object	existence status	created in superior, unchanged	The object exists in the superior workspace, and has not been destroyed in this workspace.
		created in superior, destroyed	The object exists in the superior workspace, but has been destroyed in this workspace.
		destroyed in superior, unchanged	The object formerly existed in the superior workspace, was destroyed in that workspace, and has not been restored in this workspace.
		destroyed in superior, restored	The object formerly existed in the superior workspace, was destroyed in that workspace, but has been restored in this workspace.
		not in superior, created	The object was created in this workspace.
		not in superior, destroyed	The object was created then destroyed in this workspace.
	value status	same	No slot of object has been modified in this workspace.
		different	Some slot of object has been modified in this workspace.

Table 5.1: Extended Objects

object unit	control attribute	value	meaning
base object (cont.)	time–stamp	some Co–DBMS time	The time at which the slots of the object were most recently updated.
basic slot	value status	same	The value of the slot was not changed in this workspace.
		different	The value of the slot was changed in this workspace.
	time–stamp	some Co–DBMS time	The most recent time at which the slot was changed.
sub–object slot	value status	same	No slot of sub–object was not changed in this workspace.
		different	Some slot of sub–object was changed in this workspace.
	time–stamp	some Co–DBMS time	The most recent time at which slots were changed.
reference slot	value status	same	The reference in this slot was not changed in this workspace.
		different	The reference in this slot was changed in this workspace.
	time–stamp	some Co–DBMS time	The most recent time at which the slot was changed.
set–valued slots	time–stamp	some Co–DBMS time	The most recent time a set member was created, destroyed or updated.

Table 5.1: Extended Objects (cont.)

object unit	control attribute	value	meaning
member of set–valued slot	existence status	created in superior, unchanged	The set member exists in the superior workspace, and has not been destroyed in this workspace.
		created in superior, destroyed	The set member exists in the superior workspace, but has been destroyed in this workspace.
		destroyed in superior, unchanged	The set member formerly existed in the superior workspace, was destroyed in that workspace, and has not been restored in this workspace.
		destroyed in superior, restored	The set member formerly existed in the superior workspace, was destroyed in that workspace, but has been restored in this workspace.
		not in superior, created	The set member was created in this workspace.
		not in superior, destroyed	The set member was created then destroyed in this workspace.
	other control attributes appropriate to the type of the member, as described in this table	described in this table	Presented in this table.
derived direct slot	attributes appropriate to the type of slot, as in this table	described in this table	Presented in this table.

Table 5.1: Extended Objects (cont.)

object unit	control attributes	value	meaning
derived external slot	validity status	invalid	Value of slot is not current and must be recomputed.
		valid	Value of slot is current.
	invalidated	true	The slot has been invalidated since the workspace was last committed.
		false	The slot has not been invalidated since the workspace was last committed.
	validated	true	The slot has been recomputed since the workspace was last committed.
		false	The slot has not been recomputed since the workspace was last committed.
	time−stamp	some Co−DBMS time	If (validity status = invalid) then The earliest time that the slot was made invalid since it was last made valid. else The most recent time that the slot was made valid.
	attributes appropriate to the type of the derived external slot, as noted in this table	described in this table	Presented in this table.

Table 5.1: Extended Objects (cont.)

The time-stamps in the extended objects assume the value t of the Co-DBMS timer at the time when the DOM processes an update request by an agent. There is enough information in extended objects without the time-stamps to enable the DOM to infer the update delta. Time-stamps are used for another reason: by comparing the time-stamp of a derived external slot to the time-stamps of the slots from which it is computed, it is possible for the dependency checker in the AOM (presented in Section 6.2.7) to determine which slots were changed and caused a derived external slot to be invalid; this can potentially save a great deal of effort in recomputing the derived external slot. Time-stamps are also used to ensure that an agent keeps a consistent view of objects in its cache; this is explained in Section 5.2.8.

Commit(AgentID, WorkspaceID)

The Commit operation passes all updates to workspace W up to $Superior(W)$ so that they are visible at a higher level in the hierarchy. The root workspace W_{root} has no superior and cannot be committed. A workspace $W \neq W_{root}$ can be committed only if it satisfies specific **correctness criteria**. The correctness criteria for committing a workspace are:

1. the constraint specifications of $Superior(W)$ are met by all objects in W;
2. there are no unresolved collisions in W; the existence of unresolved collisions indicates a problem that has not been resolved; preventing W from committing in this situation will block the propagation of errors to more public workspaces.

Abort(AgentID, WorkspaceID)

Aborting a workspace implies discarding all new and modified objects in that workspace and all its sub-workspaces (if exists). A workspace W can be aborted only if the following two conditions are met:

1. there are no agents which have W or a sub-workspace selected;

2. there are no uncommitted updates in any sub-workspace of W.

In other words, before aborting a workspace, all its sub-workspaces have to be recursively aborted first.

It is to be noted that, aborting W has no effect on $Superior(W)$. After the abort, both W and $Superior(W)$ offer the same view of objects.

Sometimes aborting a workspace may have a rather drastic consequences. A less costly way to undo selected updates to a workspace is to employ *compensating updates* to achieve a desired state of objects [21, 40].

5.2.8 Object check-out and check-in

A workspace can be thought of as the working area for a long open-ended transaction. The DOM permits more than one agent, possibly under the control of multiple users, to share updates to the same object in the same workspace.

Before an agent can access an object, it must check-out that object. Different check-out modes may be considered. In this work, we maintain that an object can be checked-out for either *read* or *update* access. The DOM dramatically increases the potential for concurrency and cooperation in the environment; it offers mechanisms which enable check-out of objects for update without resorting to the use

of exclusive access. Neither check-out for read nor check-out for update excludes check-out by other agents.

When an agent checks-out an object for read, the object's current state is returned. The agent places all objects it checks-out in its cache of objects (agent context or agent workspace). Consider an object X checked-out by agent A, then upon A's request, the DOM will send notifications of any updates made to X to agent A until A checks-in X.

When an agent requests to check-out an object X for update, it implicitly checks-out all *dependents*(X) for update access. In order to maintain a consistent view of objects, some conditions must be imposed on the circumstances in which an agent is permitted to check-out an object for update.

Conditions for checking-out objects for update

An agent which has selected workspace W can check-out object X for update (and implicitly *dependents*(X)) only if the following two conditions are satisfied for every $Y \in objectgroup(X)$.

- **Condition 1:**

  Y is not checked-out for update except in workspace W. This guarantees the invariant that if *dependson*(X, Y) and both X and Y are checked-out for update, then they are checked-out in the same workspace.

- **Condition 2:**

  There are no uncommitted updates to Y in any workspace W' unless $W \leq W'$. This guarantees the invariant that if *dependson*(X, Y), there are uncommitted updates to either X or Y in some workspace, and either X or Y

is checked-out for update, then the workspace with uncommitted updates is the same workspace where X or Y is checked-out or is a super-workspace of that workspace.

Since $X \in objectgroup(X)$, condition 1 implies that an object can be checked-out for update in at most one workspace. If there is a need for two agents to update X at the same time, they must check-out X in the same workspace. Condition 2 implies that an object can be checked-out only in the same workspace or in a sub-workspace wherein there are uncommitted updates to the object.

Suppose an agent submits updates $\Delta U =< u_1, u_2, ..., u_n >$ to workspace W' at time t_{Update}. Just prior to t_{Update}, at time t^*, for every workspace $W \neq W_{root}$, there exists some update delta ΔU_W such that:

$$V_W(t^*) = V_{Superior(W)}(t^*) + \Delta U_W(t^*).$$

Invariants maintained by conditions 1 and 2 guarantee that for each workspace $W \leq W'$:

$V_W(t_{Update}) = V_{Superior(W)}(t_{Update}) + \Delta U_W(t_{Update})$, where $\Delta U_W(t_{Update}) = \Delta U_W(t^*) + \Delta U$.

This result holds whether the update delta ΔU comes from committing an inferior workspace or from an agent.

Without these two conditions the DOM would have to "merge" updates to X in W with the state of X or the state of its dependents in sub-workspaces, rather than merely apply the updates. The DOM is unable to merge updates, because

this would require that it understand the semantics of the data and the intent of the agent in making the update.

When an agent updates an object to reference another, it does so within its object cache, then sometime later commits that reference to the workspace selected by the agent. The DOM must be aware of an agent's intention to update an object to reference another, so that it can ensure that these conditions are enforced should the agent commit its updates. The way this is done is presented in Section 5.2.9.

Update notifications

An agent checks-out into some workspace, and caches within its object cache, some objects that it needs to work with. The DOM sends the agent update notifications of changes to all objects checked-out so that it can keep its cache consistent with the Co-DBMS.

Each update notification contains the following information:

1. the AgentID of the agent that submitted the update and caused the notification to be sent;
2. the BaseObjectID of the base object updated;
3. the update operation which was applied to the base object;
4. a time-stamp that records the time when the update was performed.

Update notifications are distinguished as either *immediate* or *deferred*. An agent may defer handling those notifications so that to keep the user's current

view of objects from changing unexpectedly. In this case, the updates will be handled later, in the meantime, the view presented will be consistent, although somewhat out-dated.

Now, consider an agent that checked-out and cached an object, while at the same time it is deferring the handling of notifications. Suppose the time the object was last updated is more recent than the last notification handled by the agent. Then the state of objects in the agent's object cache has become inconsistent, since updates to some objects already in the cache have not been incorporated; the object just checked-out, however, have the most recent updates applied to it.

To prevent inconsistencies from occurring, the following condition is imposed:

an agent wishing to defer handling notifications it receives may do so provided that it handles all pending notifications sent before the time of the last update to the additional objects it requests to check-out.

This may be implemented as follows. When an agent requests to check-out an object, it submits the time-stamp of the last update notification handled. If the object to be checked-out has an update time-stamp (time for most recent update) greater than the value of the time-stamp sent by the agent, the check-out request is rejected. In this case the agent must handle additional notifications and re-submits the request if it so chooses.

CheckOutForRead(AgentID, BaseObjectID, LastNotificationHandled)
return extendedObject/ handleNotifications

CheckOutForUpdate(AgentID, BaseObjectID, LastNotificationHandled)

return extendedObject(s)/ handleNotifications

An agent invokes CheckOutForRead or CheckOutForUpdate to check-out an object for read or update access, respectively. The DOM returns a copy of the object or, in the case of CheckOutForUpdate, checks-out and returns all dependents of that object.

CheckIn(AgentID, BaseObjectID, LastNotificationHandled)

return ok/ handleNotifications

An agent invokes CheckIn to inform the DOM that it no longer needs to access an object.

5.2.9 Managing object references

An agent may update objects in its cache to include references to other objects (not necessarily in its cache). In some situations, the DOM may not approve of such references. For example, if the referenced object is checked-out for update in some other workspace. Therefore, to ensure the correct behavior of the system, it must be kept aware of references from one object to another in an agent's object cache that has not yet been committed to a workspace. This is achieved by having each agent inform the DOM when it updates an object so that it references, or no longer references, another object.

AddReference(AgentID, ObjectID, ReferencedObjectID)

return ok/ notAllowed

An agent informs the DOM that it has updated an object in its cache to reference another object by invoking AddReference and identifying the referencing and referenced object. This has the side effect of incrementing the reference count from the referencing to the referenced object. Suppose an agent has selected workspace W. This request will fail if the referenced object is either checked-out for update in some workspace other than W or has uncommitted changes in any workspace W' except where $W \leq W'$.

RemoveReference(AgentID, ObjectID, ReferencedObjectID)

An agent informs the DOM that it has updated an object in its cache to no longer reference another object by invoking RemoveReference and identifying the referencing and referenced object. If the referenced object is no longer referenced by any object in the agent's cache, then it is free to be checked-out for update in workspaces other than the workspace selected by the agent, subject to the conditions on checking-out an object for update presented in Section 5.2.7.

5.2.10 Updating objects in workspaces

Each agent is free to submit a batch of updates to the workspace it has selected at any time. The batch is termed an *update step*. Updates in an update step are applied atomically. Following an agent's update step, the DOM sends update notifications to every agent that has requested to be notified of updates to objects that might affect the objects it has checked-out in this or sub-workspaces. Allowing agents in sub-workspaces to receive update notifications from agents in their super-workspaces, has the advantage of providing the former agents with up-to-date changes in the state of super-workspaces, and therefore they can always base their work on most recent information.

Now consider the following scenario. Assume two agents A_i and A_j working in the same workspace, sharing updates on object X with state S. Both agents initialize their state based on S. Agent A_i applies update u_i to its copy of X, yielding $S_i = S + \Delta U_i$, and A_j applies update u_j to its copy of X, yielding $S_j = S + \Delta U_j$. Suppose that A_i and A_j now submit u_i and u_j, and the DOM sends relevant notifications to A_j.

Suppose that, due to network delays, update u_j now arrives from A_j. The DOM will not know whether A_j received the notification before or after submitting u_j. If A_j submitted u_j before handling the notification, then u_j would be invalid, since A_j may have based its update u_j on X having state S rather than S_i as it does now. On the other hand, if A_j did handle the notification, then u_j should be applied to X.

To solve this problem, the DOM must know the relative order of notifications sent and updates received. This could be accomplished using the following protocol. Before sending a notification, the DOM attaches a time-stamp to it. The DOM records the most recent time-stamp of the notifications sent to each agent. When submitting an update request, the agent also sends to the DOM, the time-stamp of the most recent notification it handled. The DOM then compares that time-stamp with that of the last update notification sent to the agent. If an update notification has been sent since the last notification processed, then the DOM knows that the agent based its update request upon incomplete information. In this case, the DOM notifies the agent that its update might be invalid and that it needs to process additional notifications. In response, the agent must handle the notifications sent and re-submits the update request if it so chooses.

This situation may recur; by the time the agent submits its request, there may be additional notifications that it must handle before the DOM accepts its request.

UpdateWorkspace(AgentID, *list* Update, NotificationTimeStamp)
return time/ handleNotifications/ invalidConstraints

An agent commits its updates to the workspace it has selected by calling UpdateWorkspace. If the request succeeds, the Co-DBMS returns the current time. The agent uses the current Co-DBMS time to alter time-stamps in its extended object cache, as explained in Section 6.2.4. When an agent commits its changes to the Co-DBMS, the Co-DBMS decrements counts of uncommitted references between objects. UpdateWorkspace will fail either if the agent has not handled all the update notifications sent by the DOM or if accepting the updates would cause one or more of the workspace constraint specifications not to be satisfied.

5.3 Rules Maintained by the DOM

All modules within the DOM work together to jointly provide a collection of services to agents. Guaranteeing internal consistency and correct operations of the DOM require that it maintain a number of rules. This section outlines those rules.

1. Unresolved collisions restrict workspace commit

If there are unresolved collisions in a workspace W, the DOM prevents W from committing to its superior workspace $Superior(W)$ so that potentially erroneous updates are confined to W.

Rule 1:

Workspaces with unresolved collisions cannot be committed.

The DOM enforces this rule by first checking for unresolved collisions within a workspace before honoring a request to commit a workspace.

2. Constraint specifications are met

Constraint specifications provide the degree of consistency which is to be maintained at all times within a workspace. The DOM guarantees:

Rule 2:

The constraint specifications which are attached to workspaces are met at all times.

The DOM preserves this rule by rejecting updates to a workspace W, either from agents which have selected the workspace or from a committing inferior workspace of W, in which one or more constraint specifications attached to W is not true.

3. Control information provided is sufficient for commit

An agent can request that updates to a workspace be committed to its superior workspace at any time. When it does so, the DOM scans the extended object cache, interprets the control information to determine how the objects were updated, and generates a list of updates which are then applied to the superior workspace. The rule which makes this possible is:

Rule 3:

The update delta can be computed at any time from the control attributes associated with objects.

This rule is maintained by the DOM because the DOM alters the control attributes on objects, as well as the values of object slots, whenever it processes an update request from an agent or from an inferior workspace. The control attributes in a workspace after each update reflect how the states of objects differ between that workspace and the superior workspace.

4. Unhandled notifications restrict updates by agents

When an agent commits updates on objects in its object cache to a workspace in the Co-DBMS, that agent does so based upon the state of the objects it had cached earlier. If the agent has failed to process all update notifications from the DOM, it may erroneously attempt to perform an update based upon stale data. The DOM guarantees:

Rule 4:
Update requests from agents will be honored only if the agent has handled all update notifications which have been sent to it.

This rule is preserved by virtue of the protocol used between the agent and the DOM which is based on time-stamps for notifications. The protocol is explained in Section 5.2.8.

It is important to note that this rule does not guarantee that an agent has responded in a proper fashion to the notifications it has received. Such a guarantee of the agent's behavior cannot, in general, be enforced by the DOM.

5. Referential integrity is enforced

The DOM enforces referential integrity within the object store:

Rule 5:

An object cannot be destroyed if there are any references to it from other objects.

The DOM maintains this rule by first checking whether an object X is referenced by any other object before honoring a request from an agent to destroy X. This includes checking whether X is referenced by another object within the object cache of any agent. The DOM knows which objects reference other objects within caches of agents, because an agent must notify the DOM when it adds or removes an object reference within its cache.

6. Workspace related to superior by update delta

Agents submit batches of updates to workspaces in the Co-DBMS. In addition, updates in workspaces may be committed to their superior workspace. The DOM guarantees:

Rule 6:

Let $V_W(t)$ be the view of objects at time t in workspace $W \neq W_{root}$, and $V_{Superior(W)}(t)$ be the view of objects in the superior workspace of W at time t.

Then $V_W(t) = V_{Superior(W)}(t) + \Delta U(t)$, where ΔU is the update delta which represents the uncommitted updates to objects in W. The update delta is a concatenation of (1) all updates which have been applied to objects in W by the agents, and (2) updates to objects in W resulting from inferior workspaces of W having committed to W.

In order to preserve this rule, the DOM enforces the two conditions on when objects may be checked-out for update by agents; these restrictions ensure that updates applied to a workspace have no effect on the states of objects in sub-workspaces. These conditions are presented in Section 5.2.8.

Chapter 6

Agents

An **agent** consists of an application (i.e., application code, along with application-specific data structures), plus the Application Object Manager (AOM). This chapter presents the architecture of an agent, describes the functionality added by the AOM, presents the interface between an application and the AOM, and summarizes the rules maintained by the AOM.

6.1 Architecture of an Agent

The application implements a particular functionality, and is what distinguishes one agent from another. The AOM consists of six modules: timer, object cache manager, out-date propagator, update notification manager, update focus handler, and dependency checker. These modules work together and offer a collection of services to the application. See Figure 6.1. Note that an application does not communicate directly with the object store; it performs updates only through the AOM, which then communicates with the Co-DBMS.

The AOM is linked with the application to create an agent; because of this,

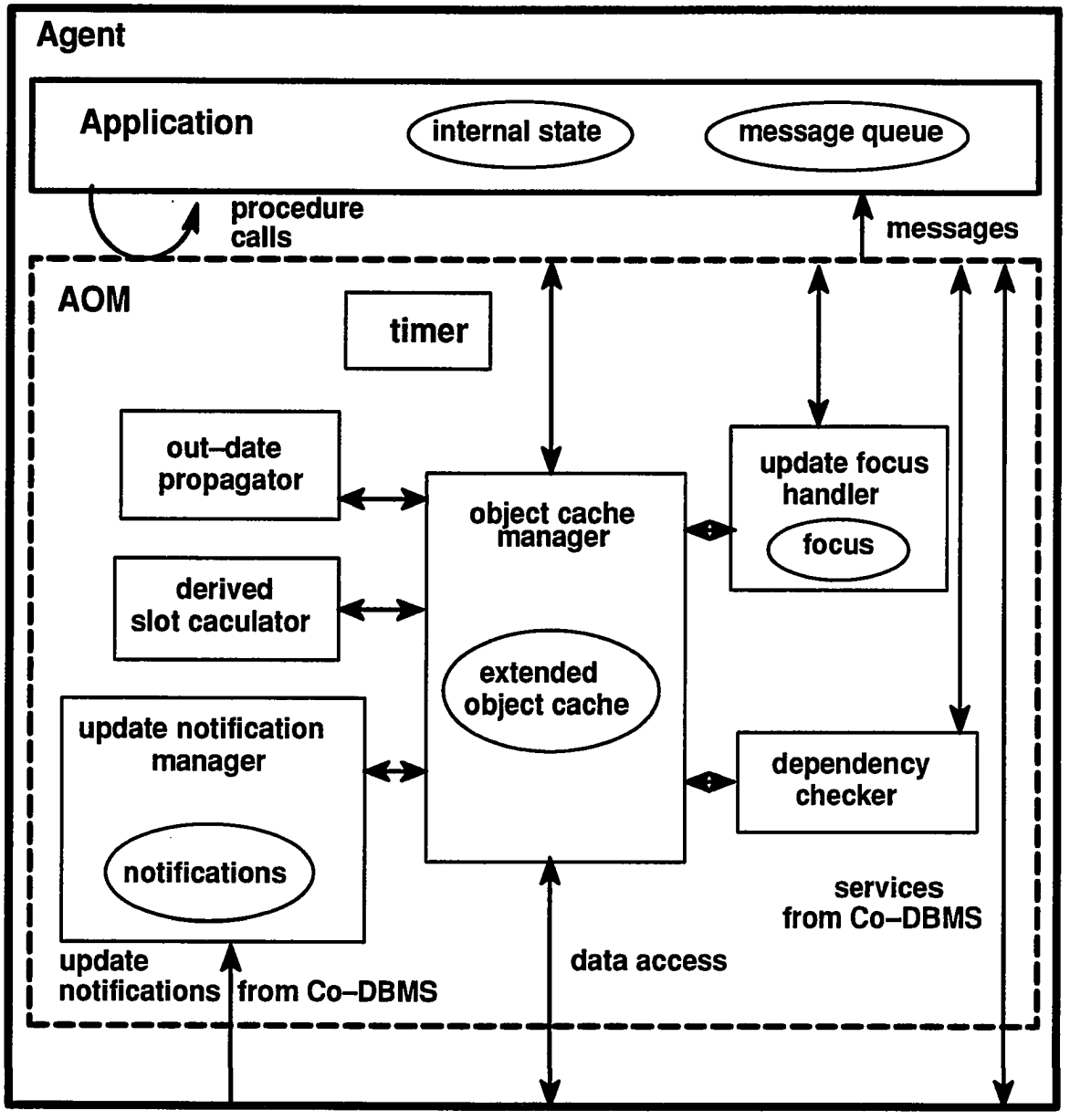

Figure 6.1: Architecture of an Agent

communication between the application and the AOM is inexpensive. Thus, frequent interaction between the application and the AOM is not inefficient, and the granule of interaction can be small – operations can involve slots of base objects, as compared to entire objects, which is the case between agents and the Co-DBMS.

The agent timer

Each agent maintains an integer-valued timer. The timer in each agent is separate from, and run asynchronously with respect to, the timer in the Co-DBMS and the timers in other agents. The timer represents the amount of time elapsed since an agent last committed its updates to the workspace it has selected. The timer is initialized to zero when an agent selects a workspace. It is incremented whenever the AOM processes any update request from the application. The timer is reset whenever the agent commits its changes. The times at which slots of objects change value are stored in the extended object cache, which holds the same information as does the extended object store in the Co-DBMS.

Message queue

One of the responsibilities of the AOM is to ensure that an application is made aware of the occurrence of asynchronous events such as an update to a shared object or a change in work status. The AOM notifies the application of an event by creating a message with a time-stamp that indicates what events occurred and appending the message to the application's **message queue**.

Object cache manager

Within each agent there is a cache of extended objects which the agent has checked-out and is currently accessing and manipulating. The cache is similar to a workspace in the Co-DBMS in that all updates are encapsulated within the cache; the updates are applied atomically to the workspace the agent has selected when the agent commits its changes. The use of a cache enables an agent to make experimental updates to local copies of objects. Unlike workspaces in the Co-DBMS, the lifetime of the extended object cache is tied to that of the agent. The **object cache manager** within the AOM gives an application access to the extended object cache by handling requests both to load data not yet cached and to update the cache.

6.2 Functionality of the Application Object Manager

This section describes each service that the AOM offers to applications, presents the programmatic interface which an application uses to access the service, and explains how modules within the AOM operate in order to provide that service.

6.2.1 Services from the Co-DBMS

Some of the services available to an application are slightly modified versions of services from the Co-DBMS which are passed up through the AOM to an application. This section describes those services.

ApplicationBegin(UserName, ApplicationName)

When an application begins operation it must notify the AOM. It does so

by calling ApplicationBegin. In response the AOM initializes itself and registers the agent with the Co-DBMS by calling ConnectAgent. The AOM remembers the AgentID returned by ConnectAgent for use in subsequent requests to the Co-DBMS.

ApplicationEnd()

return ok/ workspaceSelected

An application must also notify the AOM when it wishes to end operation. It does so by invoking ApplicationEnd. The AOM permits termination only if the application has no workspace currently selected. The AOM calls DisconnectAgent in the Co-DBMS when the application ends.

The AOM contains procedures that provide the following framework services to an application.

- creating and destroying workspaces
- workspace selection
- constraint specifications
- committing and aborting workspaces
- collision recording
- monitoring work status

These procedures provide the same functionality that the Co-DBMS offers to agents, as described in Chapter 5. Invoking any of them calls the Co-DBMS procedure of the same name.

6.2.2 Object check-out and check-in

An application requests that an object be cached by checking-out that object. If the application needs only read access, it should check-out the object for read; if it needs update access it must check-out the object for update. While an object is checked-out, the Co-DBMS will send asynchronous notifications of updates made by other agents to that object to the **update notification manager** in order that the cache be made consistent; the way this is done is explained in Section 6.2.5. An object will remain cached and accessible to the application until the application checks-in the object.

ApplicationCheckOutForRead(ObjectID, LastMessageHandled)
return extendedObject/ handleMessages

ApplicationCheckOutForUpdate(ObjectID, LastMessageHandled)
return extendedObject(s)/ handleMessages

An application calls ApplicationCheckOutForRead or ApplicationCheckOutForUpdate to check-out an object for read or update access, respectively. The AOM calls CheckOutForRead or CheckOutForUpdate in the Co-DBMS, respectively, and returns to the application a copy of the extended object or, in the case of CheckOutForUpdate, checks-out and returns all dependencies of that object.

The same restrictions apply on an agent in checking-out an object for update, as presented in Section 5.2.8. There is one difference between ApplicationCheckOutForRead (or ApplicationCheckOutForUpdate) offered by the AOM to an application and CheckOutForRead (or CheckOutForUpdate, respectively) offered by the Co-DBMS to agents: an application passes the time-stamp of the last message

it handled to ApplicationCheckOutForRead (or ApplicationCheckOutForUpdate); the AOM converts this time-stamp to the time-stamp of the last notification handled by the agent and passes that to CheckOutForRead (or CheckOutForUpdate, respectively).

ApplicationCheckIn(ObjectID, LastMessageHandled)
return ok/ uncommittedUpdates/ handleMessages

An application invokes ApplicationCheckIn to inform the AOM that it no longer needs to access an object. The AOM calls CheckIn in the Co-DBMS so that it will then send no more notifications to the agent of updates to that object. ApplicationCheckIn will fail if the application has failed to handle all messages sent to it, or if, in the case of an object checked-out for update, there are uncommitted updates to the object in the cache.

6.2.3 Reading objects in cache

An application must be able to read the contents of objects in order to initialize and keep current its internal data structures. The object cache manager gives applications a programmatic interface to access objects in the cache.

ReadSlotValue(OID, SlotName)
return value

The value returned by the ReadSlotValue depends on the type of the slot. See Table 6.1.

object unit	**value returned by *ReadSlotValue***
basic slot	value of slot of appropriate type, as described by the schema
sub–object	OID of the sub–object
set–valued slot	values (or OID in case of sub–object) of members of the set
object reference slot	OID of the referenced object
derived direct slot	value of slot of appropriate type, as described by the schema
derived external slot	the value of the slot if it is valid, else (undefined) if the slot is invalid

Table 6.1: Reading Values of Objects in Cache

6.2.4 Updating objects in cache

An application modifies data by updating objects in cache, then committing those updates to the workspace it has selected. This section discusses how derived external slots are marked invalid, presents the programmatic interface with which an application updates cached objects, and describes what effect each update operation has on the extended object cache and on other components of the AOM.

Invalidating derived external slots

When slots in a base object X are updated, derived external slots in X, and in other objects that reference X, may be invalidated. It is unreasonable to assume that every application conscientiously invalidates derived external slots whenever it updates slots in the object cache which may affect the derived external slots. Furthermore, updates to the extended object cache by the update notification manager (described in Section 6.2.5) may affect the validity of derived external slots.

For these reasons, the out-date propagator in the AOM performs the task of invalidating derived external slots in the object cache whenever slots upon which they depend, as defined by the schema, are updated by either the application or the update notification manager. The AOM does not know how to recompute the new value of the derived external slot. Instead , it is the responsibility of applications, and may require an arbitrary amount of computation.

One update may have a ripple effect in which derived external slots, and other derived external slots affected by those slots, are affected. The out-date propagator recursively marks derived external slots affected by the update as **invalid**. The

out-date propagator is guaranteed to have update access to all objects affected, since when X was checked-out for update so were *dependents*(X).

When an application calls an update procedure, the out-date propagator completes its task before the call returns control to the application. Thus, out-date propagation occurs synchronously with respect to update requests from the application.

Recomputing derived direct slots

Just as the value of derived external slots can become invalid when a slot upon which it depends has been updated, so can a value of a derived direct slot. Instead of calling the out-date propagator merely to mark the derived direct slot out-dated as it does to a derived external slot, the AOM invokes the **derived slot calculator** to recompute the value of the derived direct slot. The derived slot calculator computes the value of the derived direct slot based upon the specifications in the schema.

Update procedures

An application updates an object by calling procedures in the object cache manager. When an application updates an object, the agent timer is incremented; the time that an update occurs is used to time-stamp updated slots.

Table 6.2 shows the update procedures that can be called for each type of slot in an object and what effect each procedure has on the extended object cache. Control attributes are used when an agent commits its updates to the Co-DBMS

object unit	update procedure	modifications in extended object cache
basic slot	UpdateValue	value := new value value status := different time–stamp := t_{Update}
set–valued slot	CreateMember	existence status := not in workspace, created time–stamp := t_{Update}
	DestroyMember	If (existence status = created in workspace, unchanged) then existence status := created in workspace, destroyed else if (existence status = destroyed in workspace, restored) then existence status := destroyed in workspace, unchanged else if (existence status := not in workspace, created) then existence status := not in workspace, destroyed time–stamp := t_{Update} DestroyMember calls RemoveReference for each uncommitted inter–object reference that is removed as a result of destroying the set member.
	RestoreMember	If (existence status = created in workspace, destroyed) then existence status := created in workspace, unchanged else if (existence status = destroyed in workspace, unchanged) then existence status := destroyed in workspace, restored else if (existence status := not in workspace, destroyed) then existence status := not in workspace, created time–stamp := t_{Update} RestoreMember calls AddReference for each uncommitted inter–object reference that is added as a result of restoring the set member.

Table 6.2: Procedures to Update Objects

object unit	update procedure	modification to extended object cache
object reference slot	UpdateReference	value := OID of object to reference or value := null value status := different time–stamp := t_{Update} When an application adds a reference from one object to another, the AOM in the agent calls AddReference in the Co–DBMS. When an application removes a reference, the AOM calls RemoveReference in the Co–DBMS.
derived external slot	SetValid	validity status := valid validated := true time–stamp := t_{Update} SetValid is called by an application after it recomputes and updates the value of the derived external slot.
	SetInvalid	if (validity status = valid) then validity status := invalid time–stamp := t_{Update} invalidated := true SetInvalid is called recursively by the out–date propagator.
	procedures to update value of derived external slot	The procedures, in this table, that can be invoked to update the value of a derived external slot depend on the type of the value of the derived external slot, as specified by the schema. These procedures are called by an application that has recomputed the value of the slot.
derived direct slot	procedures to update value of derived direct slot	The procedures, in this table, that can be invoked to update the value of a derived direct slot depend on the type of the value of the derived direct slot, as specified by the schema. These procedures are called only by the derived slot calculator and not by the application. To an application, the value of a derived direct slot is always current.

Table 6.2: Procedures to Update Objects (cont.)

in order to determine the update delta between the object cache in the agent and the workspace which the agent has selected.

Four side-effects of every update exist:

1. control attributes of both the object that contains the updated slot, and every object that owns that object are updated as follows:

   value status := *different*

   time-stamp := t_{Update};

2. the update will cause derived external slots affected by the update to be invalidated by the out-date propagator;

3. the update will cause derived direct slots affected by the update to be recomputed by the derived slot calculator;

4. the update focus handler will deliver a message to the application if the update matches an interest placed by the application. (The operation of the update focus handler is explained in detail in Section 6.2.6.)

The AOM enforces two conditions on when an application can update an object in the object cache:

1. the application must have checked-out the object for update;

2. the application must have handled all messages sent to it by the update focus handler; this is done using a protocol similar to that explained in Section 5.2.8.

6.2.5 Handling update notifications

When an agent commits updates on object X to workspace W, the object cache within every other agent that has selected either W or a sub-workspace of W and which has checked-out X will become stale. The update monitor in the Co-DBMS guarantees that each agent that is checking-out X will be sent asynchronous notifications of all updates to X.

When a notification of an update by another agent is received from the update monitor, that update must be incorporated or "merged" into the agent's object cache. This action is performed by the update notification manager in the AOM. Like the out-date propagator and the derived slot calculator, the update notification manager operates automatically on behalf of the application.

It is important to note that most existing systems with notification capabilities are limited to notifying human users about the status of shared objects. They assume that only the human user is active. A cooperative environment, however, should have active components in the sense that it be able to monitor the activities in the database and automatically perform some operations in response to changes made to database objects [43, 6]. Our work provides the update monitor and the work status monitor in the Co-DBMS, and the out-date propagator, the derived slot calculator, and the update notification manager in the agent to serve these purposes.

Purpose of the update notification manager

Suppose agent A has selected workspace W. Let $V_A(t)$ represent the view of the object cache within agent A at time t, and $V_W(t)$ represent the view of objects in workspace W at time t, to the extent that update notifications have been merged

into the object cache in A.

Rule: $V_A(t) = V_W(t) + \Delta U(t)$, where $\Delta U(t)$ is the update delta from W to A. The update delta represents the uncommitted updates on the object cache performed by the application.

Suppose notification of update u is sent to the update notification manager, and that the update notification manager merges u into the object cache at time t_{Merge}. It does so by altering V_A to reflect update u and computing a new update delta which is as close as possible to the old one. That is, the update notification manager restores the rule by finding some small δU such that:

$$V_A(t_{Merge}) = V_W(t_{Merge}) + \Delta U(t_{Merge}),$$

$$V_W(t_{Merge}) = V_W(t_{Merge} - 1) + u, \text{ and}$$

$$\Delta U(t_{Merge}) = \Delta U(t_{Merge} - 1) + \delta U$$

Operation of the update notification manager

When an update notification manager receives an update notification from the update monitor, it attempts to reflect the change in the object cache. Table 6.3 shows how the update notification manager updates the object cache for each type of update notification it can receive. As in the case with updates from an application, updates from the update notification manager can have side effects.

It is important to note that the manner in which the update notification man-

object unit	update notification	modifications to extended object cache
basic slot	UpdateValue to v	value := v value status := same time–stamp := t_{Update} An update notification may describe an update to a slot in an object that was a member of some set but that the application has destroyed. In this case, the update notification manager will first call RestoreMember to restore the object then perform the update.
set–valued slot	CreateMember	existence status := created in workspace, unchanged time–stamp := t_{Update}
	DestroyMember	existence status := destroyed in workspace, unchanged time–stamp := t_{Update} The update notification manager calls RemoveReference for each uncommitted inter–object reference that is removed as a result of destroying a set member.
	RestoreMember	existence status := created in workspace, unchanged time–stamp := t_{Update} The update notification manager calls AddReference for each uncommitted reference that is added as a result of restoring a set member.
reference slot	UpdateReference to OID or null	value := OID or null value status := same time–stamp := t_{Update} When the update notification manager removes an existing reference from one object to another, it calls RemoveReference in the Co–DBMS.

Table 6.3: Handling Update Notifications

object unit	update notification	modifications to extended object cache
derived external slot	SetValid	if (invalidated = false) then validity status := valid time–stamp := t_{Update} else validity status := invalid If an agent A recomputes and sets a derived external slot as valid, that validity can propagate to the object cache of another agent A only if A has never invalidated derived external slot in its cache.
	SetInvalid	When the update notification manager incorporates other agents' updates into the object cache, the out–date propagator will automatically invalidate any derived external slots affected; no action need to be taken by the update notification manager when it receives notifications of a derived external slot having been marked invalid.
	updates to the value of the derived external slot	After an agent recomputes a derived external slot, it updates the slot to contain the new value. Thus the update notification manager may receive notifications of updates by other agents to a derived external slot. It responds by applying those updates, as described by this table, to the derived external slot in the object cache.
derived direct slot	updates to the value of the derived direct slot	When the update notification manager incorporates other agents' updates into the object cache, the derived slot calculator will automatically invalidate any derived external slots affected; no action need be taken by the update notification manager when it receives notification of a derived direct slot having been updated.

Table 6.3: Handling Update Notifications (cont.)

ager merges updates from other agents into the object cache is **syntactic** rather than **semantic**. The update notification manager does not understand any meaning which may be assigned to the state of objects. Thus, when the update notification manager merges updates it may unknowingly undo updates to or adversely affect the state of the object cache within the agent. In such a case, the application is responsible for applying compensating updates to the object cache in order to restore it to a "semantically consistent" state before committing the state of the object cache to a workspace in the Co-DBMS.

Deferred handling of updates

In normal operation, the update notification manager makes asynchronous changes to the object cache in response to update notifications, received from the update monitor in the Co-DBMS, that describe updates made by other agents. Thus, the view of data presented to an application is subject to change. At times, it may be convenient for an application to present a static view of objects to a product developer, and therefore, the processing of update notifications by the update notification manager is to be deferred. For example, a product developer might choose not to be bothered by updates made by other product developers until the end of each day. Note that the disadvantage of deferring the incorporation of updates made by other product developers is that the product developer will not be aware of potentially conflicting or erroneous updates until the merging of updates resumes. But at that time other updates may have been predicated on the erroneous updates, and correcting the resulting problem will be more difficult. In general, identifying conflicts early than late in the process reduces the cost.

The AOM offers applications the ability to cause the update notification manager to defer or to resume the merging of update notifications into the object cache.

For the period of time that the merging is suspended, the view of objects that the application sees may be out-of-date.

When the object cache is stale, an application operates based on the view of the world that is somewhat incorrect. For this reason, the application is restricted in what it can do while the update notification manager has been turned off; in particular, it is not allowed to update objects in the Co-DBMS. The handshaking used in the procedures CheckOutForRead, CheckOutForUpdate, CheckIn, ApplicationCheckOutForRead, ApplicationCheckOutForUpdate, ApplicationCheckIn, and ApplicationCommit prevent an application from checking objects out or in or from committing its updates to the Co-DBMS unless the update notification manager has handled all update notifications and the application has handled all the resulting messages.

DeferUpdateHandling()

An application calls DeferUpdateHandling when it wants to suspend operation of the update notification manager. The application is then assured that any changes to the objects in the cache are results of its updates, not those of other agents.

ResumeUpdateHandling()

An application calls ResumeUpdateHandling to continue operation of the update notification manager. When the update notification is running, objects in the cache are subject to change asynchronously.

6.2.6 Handling update focus

After an agent checks-out and caches an object X, the update monitor in the Co-DBMS sends notifications of any updates to X to the update notification manager in the agent, which uses the notification to make the object cache current. The application in an agent reads data from and submits updates to the object cache, then at some point commits those updates to the workspace it has selected. When the update notification manager updates the object cache, it may make changes that require the application either to adjust its internal state, or to make updates to the object cache which compensate for updates from another agent, or both. Thus, the application must be aware of some set of updates to the object cache.

Because different applications have different semantics, those updates in which an application is interested in being notified depends on the particular application. The AOM does not understand the semantics of applications. Thus, it is the responsibility of the application to inform the AOM of which updates it needs to be informed about. It does so by registering **interests** with the **update focus handler** in the AOM. Each interest identifies some set of updates. When any update specified by an interest occurs, the update focus handler sends a message to the application that describes the update. An application registers some number of interests with the update focus handler; the interests are chosen so that the set of updates in which the application is interested is covered by the interests.

Collectively, a set of interests demarcates a region of interest, referred to as **update focus**. The update focus should include updates to data upon which the application is basing its operation – the **read set**. When an application performs an update to the object cache, the update focus handler does not send a notification of that update back to the application. The update may, however, trigger

the out-date propagator or the derived slot calculator to perform further updates on derived external or derived direct slots, respectively; these updates may cause messages to be sent to the application that performed the original update if it had registered an interest in some derived slot affected.

The updates in which an application is interested may vary over time. An application is free to adjust its focus at any time by registering additional interests or unregistering an interest it had previously registered.

The mechanism of update focus and messages provided by the AOM and the Co-DBMS is what the framework offers for flexible concurrency control.

Matching updates to interests

Table 6.4 shows each interest that can be registered by an application and indicates which updates will cause a message to be sent to an application that has registered that interest.

Data-driven and demand-driven recomputation

If an application recomputes derived external slot immediately whenever the slot is invalidated, the result is **data-driven** computation, and is similar to the operation of a spreadsheet, which recomputes computed fields whenever data upon which they depend have changed. An application can also achieve **demand-driven** computation by deferring recomputation of a derived external slot until the value of that slot is needed.

interest	corresponding update
value of slot S	S is a basic slot *UpdateValue*(S,v) S contains a sub–object application would use "existence of object" or state of object described below S is set–valued *CreateMember*(S,X), *DestroyMember*(S,X), *RestoreMember*(S,X), or any update to a member of S S is a reference slot *UpdateReference*(S,OID) S is a derived direct slot the derived slot calculator updates the value of the derived direct slot S is a derived external slot *SetValid*(S) or *SetInvalid*(S)
existence of object X	*DestroyMember*(S,X), where X is a member of set–valued slot S *DestroyMember*(S,Y), where X is a sub–object of Y *RestoreMember*(S,X), where X is a member of set–valued slot S *RestoreMember*(S,Y), where X is a sub–object of Y
state of object X	any update to any slot of object X any update to any slot of a sub–object of X

Table 6.4: Update Interests and their Corresponding Updates

If CPU resources were infinite, the value of derived external slots could be constantly recomputed and there would be no need for demand-driven computation. They are not, of course, so the tradeoff between computational expense and keeping derived external slots valid, and therefore the choice between use of data- or demand-driven computation, is an engineering trade-off.

AddInterest(SpecificationOfInterest)
return InterestID

An application enlarges its focus by calling AddInterest and indicating the specification of interest to be registered. When an update occurs to the object cache that matches an interest that the application has registered, the update focus handler sends a message to the application.

RemoveInterest(InterestID)

An application calls RemoveInterest when its focus has been reduced and notification of updates corresponding to an interest which was registered earlier are no longer needed.

6.2.7 Checking update dependencies of derived external slots

When an application needs to recompute the value of an invalidated derived external slot, it may be useful to know which slots have changed since the slot was last valid. The **dependency checker** in the AOM compares time-stamps in the extended object cache to identify those slots.

When an object is cached in an agent, it contains time-stamp attributes from the Co-DBMS timer. When the update notification manager merges updates into the extended object cache, it assigns the time-stamps of the notifications to time-stamp attributes of cached objects; these time-stamps are also from the Co-DBMS timer. When an application makes an update to the extended object cache, however, the time-stamps used are from the agent timer. Thus, time-stamps in the extended object cache will be a mix of time-stamps from the Co-DBMS timer and from the agent timer. Suppose t_1 and t_2 are two time-stamps. We define a total order of time-stamps as follows:

$t_1 < t_2$ if and only if one of the following three conditions is met:

1. t_1 and t_2 are Co-DBMS time-stamps and $t_1 < t_2$;
2. t_1 and t_2 are agent time-stamps and $t_1 < t_2$;
3. t_1 is a Co-DBMS time-stamp and t_2 is an agent time-stamp.

As explained in Section 5.2.7, each slot in an object has a time-stamp attribute. Updates to the extended object cache performed by the application, out-date propagator, derived slot calculator, and update notification manager all maintain the rule if a derived external slot E is invalidated, its value depends on slot S, and S has been updated since E was last valid, then $time{-}stamp(E) \leq time{-}stamp(S)$.

The dependency checker works by comparing the time-stamp of the derived external slot E with the time-stamp of each slot S upon which it depends. If the time-stamp of E is not greater than the time-stamp of S, then S is included among those slots which, as a result of being updated, caused E to become invalid.

CheckUpdateDependency(OID, DerivedExternalSlot)

return set Slot

An application calls CheckUpdateDependencies and specifies a particular derived external slot in order to retrieve the set of slots which have changed since the derived external slot was last computed.

6.2.8 Committing updates to workspace

When an application wishes to save its updates of objects to the workspace it has selected, it *commits* to the Co-DBMS. When the application commits its updates, the AOM computes the update delta, that is, a list of updates which represent the difference between the workspace and the object cache, and submits that list by calling UpdateWorkspace, as discussed in Section 5.2.10. An application may request that its updates be discarded rather than committed; in this case the AOM reloads cached objects from the data store and reinitializes the object cache. Table 6.5 shows how the object cache manager in the AOM computes the update delta by recursively scanning each object in the object cache.

CommitUpdate(LastMessageHandled)

return ok/ handleMessages/ invalidConstraint

An application commits its updates by invoking CommitUpdate. The request will fail either if the application has not handled all the messages sent to it by the update focus handler, or if not all constraint specifications of the workspace selected by the application are true in the object cache.

object unit	the state of control attributes	update generated
basic slot	value status = different	value status := same time–stamp := time–stamp + t_{Commit} The time–stamp of each slot in the object cache that was updated is incremented by the current Co–DBMS time when the agent commits, in order to convert it from agent time to Co–DBMS time. *UpdateValue*(X,, S, value, time–stamp)
sub–object or member of set–valued slot	value status = different	value status := same time–stamp := time–stamp + t_{Commit} Recursively scan each slot in object and generate updates according to this table.
set–valued slot	Generate update for each member where: existence status = not in workspace, created	existence status := created in workspace, unchanged time–stamp := time–stamp + t_{Commit} *CreateMember*(X, S, OID of new member, time–stamp)
	Generate update for each member where: existence status = created in workspace, destroyed	existence status := destroyed in workspace, unchanged time–stamp := time–stamp + t_{Commit} *DestroyMember*(X, S, OID of member, time–stamp)
	Generate update for each member where: existence status = destroyed in workspace, restored	existence status := created in workspace, unchanged time–stamp := time–stamp + t_{Commit} *RestoreMember*(X, S, OID of member, time–stamp)

Table 6.5: Computing Differential Updates

object unit	the state of control attributes	update generated
object reference slot	value status = different	value status:= same time–stamp := time–stamp + t_{Commit} *UpdateReference*(X, S, value, time–stamp)
derived external slot	validity status = invalid and invalidated = true	invalidated := false validated := false time–stamp := time–stamp + t_{Commit} *SetInvalid*(X, S, time–stamp)
	validity status = valid and validated = true	invalidated := false validated := false time–stamp := time–stamp + t_{Commit} *SetValid*(X, S, time–stamp) Recursively scan the value of the derived external slot and generate updates according to this table
derived direct slot	value status = different	value status := same time–stamp := time–stamp + t_{Commit} Recursively scan the value of the derived direct slot and generate updates according to this table.

Table 6.5: Computing Differential Updates (cont.)

AbortUpdate(LastMessageHandled)

return ok/ handleMessages

An application discards its updates by invoking AbortUpdate.

6.3 Rules maintained by the AOM

All modules within the AOM work together to jointly provide a collection of services to an application. Guaranteeing internal consistency and correct operation of the AOM requires that it maintain a number of rules. This section summarizes the rules.

1. Unhandled messages restrict updates by applications

When an application makes updates to objects in the object cache, it does so based upon the state of the objects in its read set. If the application has registered interests in certain updates, it may receive messages from the update focus handler. The AOM guarantees:

Rule 1:

An application can update the object cache only after it has seen all messages sent to it since its last update.

This rule is preserved by virtue of the protocol used between the application and the object cache manager. Because of the protocol, the object cache manager may refuse an update request from the application.

2. Object cache related to workspace by update delta

An application caches copies of objects and performs updates on those objects. Because of updates by other agents, the cache may grow stale. The AOM guarantees:

Rule 2:

Let $V_W(t)$ be the view of objects at time t in the workspace selected by an agent and $V_A(t)$ be the view of objects in the agent's object cache at time t:

Then $V_A(t) = V_W(t) + \Delta U(t)$, where ΔU is the update delta which represents the uncommitted updates to objects in the cache. The update delta is composed of updates to the object cache by the application and updates to the workspace from other agents.

The object cache manager preserves this rule by incorporating updates from the application into the object cache. The update notification manager preserves this rule by merging updates from other agents, as described by update notifications, into the object cache and making adjustments to the update delta.

3. Control attributes provided are sufficient for commit

An application can choose to commit its updates to the Co-DBMS at any time. When it does so, the object cache manager scans the extended object cache, interprets the information provided by the control attributes to determine what updates were performed, and generates a list of updates which is then presented to the Co-DBMS. The rule that makes this possible is:

Rule 3:

The update delta can be computed at any time from the control attributes associated with objects.

The object cache manager, the out-date propagator, the update notification manager, and derived slot calculator are the only modules that update the object cache. They change the value of the control attributes in such a way that the difference between the state of objects in the cache and objects in the Co-DBMS is captured by the state of the control attributes.

4. Derived external slots are automatically invalidated

An application need not be aware of all derived external slots which may be affected by an update, because the out-date propagator guarantees:

Rule 4:

For every slot S and derived external slot E that depends on S:
If S is updated, then the validity status of E is set to invalid.

The out-date propagator behaves as a truth maintenance system by recursively marking derived external slots as invalid after each update to the object cache made by either the application or by the update notification manager.

5. Relative time-stamps of slots are maintained

The dependency checker determines for a specified derived external slot which slots upon which it depends have been updated since the derived external slot was last computed. In order to do so, the dependency checker compares the time-stamp of the derived external slot with the time-stamps of the source slots; if a derived external slot has a time-stamp that is not greater than that of a source slot, then the source slot may have been updated since the derived external slot was last

computed and should be included in the reply returned by the dependency checker.

The rule which guarantees that the algorithm in the dependency checker works is:

Rule 5:

For every slot S and derived external slot E that depends on S:

If S has been updated since E was last computed, then $time - stamp(E) \leq time - stamp(S)$.

This rule is maintained by the out-date propagator as follows:

When the out-date propagator is invalidating a derived external slot, and that slot is valid, the out-date propagator marks the slot as invalid and sets the time-stamp of slot to that of the update. If the derived external slot has already been marked invalid, the out-date propagator changes the time-stamp of the slot to the minimum of the time-stamp of the update and the time-stamp already assigned to the derived external slot.

Chapter 7

Developing Applications for Cooperative Environments

The preceding two chapters have provided an operational definition of the Co-DBMS and the agents, and have identified the operation rules maintained by the DOM and the AOM. The inclusion of the AOM and the DOM between applications and the database and containing an application to access data only through the AOM, however, do not prevent unsuitable operation of an application. This is because there are certain requirements on the application within an agent to make it behave correctly. An application that fulfills these requirements is termed **cooperative application (or co-application)**. Among co-applications there is a range of levels of cooperation of the application with the AOM; higher level of cooperation admit higher levels of concurrency.

This chapter identifies what is required of an application for it to be co-application and what minimal alterations are needed to upgrade an existing application to a co-application. The chapter also explains what it means for an application to handle messages from the focus handler, and discusses levels of

cooperation of the application with the AOM.

7.1 Cooperative Applications

The AOM and the DOM give multiple applications simultaneous update access to database objects. This places special requirements on applications so that they do not interfere with each other. An application that meets these requirements is termed **co-application**. This section defines what the requirements are.

7.1.1 Requirements of a co-application

So that an application does not interfere with updates of other applications, it must meet the following three requirements.

1. All access to objects by an application must only be through the interface provided by the AOM.

2. As the application operates and reads data from the object cache and initializes internal data structures, it must adjust its focus to include updates to all values upon which it is currently basing its internal state, so that it will receive messages from the focus handler when those values change because of updates of other applications.

3. The application must handle all messages sent to it by the focus handler before it commits its updates to the Co-DBMS. (Message handling is covered in the next section.)

Note that, according to the above definition, any application can be a co-application merely by never committing its updates. Therefore, satisfying the above requirements does not imply the *usefulness* of a co-application, but merely that it is not

harmful to updates performed by other application. If a product developer uses an application which fails to follow the above requirements, unpredictable alterations to objects in the Co-DBMS may result; in such a case not only will progress on the work will be hindered, but also damage to or reversal of the contributions of other developers may occur.

7.1.2 Converting existing applications to co-applications

This section discusses how an existing application, which was not built to be used in a cooperative environment, can be converted to a co-application and be somewhat useful, but still need not know how to handle any messages. Although such a conversion ensures that the application will not disrupt the efforts of other applications, the application will be unable to affect progress on the work in the face of concurrent access to shared objects by other applications.

Here is what is required of the application:

- upon starting, the application registers itself with the AOM;
- the application checks-out for update any objects it needs to change, and checks-out for read any other objects it needs to access;
- the application will perform its task as usual, including interacting with the user as necessary, until the task is complete; updates performed on internal data structures need to be committed first to the object cache then to the Co-DBMS;
- at commit time, the application will convert updates on internal data structures to updates on the object cache then request that they be committed to the Co-DBMS; but if any messages have arrived from the focus handler,

the application must inform the user that the updates must be aborted and the agent restarted.

The application must abort its updates if messages have been sent from the focus handler. This is because a simple-minded application that lacks the intelligence to handle messages must not predicate any updates to objects upon other objects which have changed by another application (as described by the messages). An application that operates in this fashion would offer no benefit in the face of concurrent operation. Note that this scheme is analogous to optimistic concurrency control: acquisition of the same lock by two transactions require that one transaction aborts, but if there is no such interference then both transactions can commit their updates [45].

7.2 Message Handling

Even though the update notification manager has incorporated external updates into the agent's object cache, a critical question remains: Has the internal state of the application, that is, the application's view of the world that it constructed from the object cache *before* the update notifications were received, been disturbed? The answer is – maybe. It depends on the semantics of the application; these are known only to the application itself. The best assistance that can be offered by the application is to let it inform the focus handler what objects and slots it has assumed to be static, then notify it via messages if any of those change.

The above protocol assumes that in most cases an application will be able to handle notifications it receives. This is, of course, a form of optimistic concurrency control. In the worst case, an application is unable to incorporate the changes made by another application into its view and the product developer cannot con-

tinue the current thread of updates; this is analogous to a database transaction abort. However, the stirring motivation for allowing multiple applications to update shared objects is to permit *cooperative* updates to be made. The essence of cooperation is that updates made by one product developer and his or her applications are not catastrophic to the ongoing efforts of other cooperating product developers [28, 31, 48]. This does not mean that the update might not cause problems with the functionality or correctness of the product. In general, an application, probably under the direction of its user, will try to adjust to changes by other applications so that the overall functionality or correctness of the product are retained.

As described in Section 6.2.6, each application provides its update interests to the focus handler which demarcate its focus. As a result, an update within that focus generates a message which is sent to that application. As described above, a co-application must handle every message received, if it is to commit its updates to the Co-DBMS.

A message from the focus handler provides a description of an update to the object cache that occurred as the result of another application's update. For an application to handle a message means that it make its internal representations and data structures reflect the new state of objects in the Co-DBMS. This might include updating a graphical display which offers the user a view of objects. To handle a message may also mean that the application must perform compensating updates on its object cache in order to restore semantic constraints of the database objects or to amend changes which were performed automatically by the update notification manager.

Just before an application receives a message, it presumably has been running and its controlling user has made updates to the view of objects offered by the application. The manner in which an application handles a message depends upon the application, its assumptions about the database objects before the message arrived, the focus and extent of updates which caused the message to be sent, the semantics of data involved, and the state of objects in the object cache. For this reason, the AOM cannot ascertain whether an application has appropriately handled a message from the focus handler.

The requirement that applications respond in a reasonable fashion to messages is not trivial. In the general case, handling a message may require an application to exhibit an arbitrary amount of intelligence. The amount of intelligence that an application has is called **level of cooperation**, and is discussed next.

7.3 Levels of Cooperation

In many cases an application will be able to handle a message from the focus handler by adjusting its internal data structures to incorporate the update described by the message and by performing compensating updates in order to achieve a required level of consistency. In some cases the application will be unable to do so incrementally. For example, a simulator, in response to a message that reports a change in the schematic, may be unable to modify the results of an ongoing simulation and will have to restart the simulation. In still other cases, an application may simply have not enough intelligence to enable it to handle a particular message from the focus handler.

The level of cooperation determines the variety of circumstances that an ap-

plication is able to handle messages from the focus handler and still continue to operate without aborting the updates it has made.

An application that has achieved a low level of cooperation with the AOM may frequently be forced to abort operation when other applications update objects upon which it has built its internal state. An application with a high level of cooperation with the AOM can usually continue operating even when there are updates to shared objects that are performed by other applications.

7.3.1 Low level of cooperation

If an application lacks the intelligence needed to handle messages from the focus handler, then it cannot commit its changes to the Co-DBMS if another application updates objects upon which it has built its internal state. Such a simple-minded application might merely inform its user that it must be restarted.

It is interesting to note that such a low level of cooperation corresponds to optimistic concurrency control.

7.3.2 Medium level of cooperation

Certain combinations of updates commute. For example, inserting member m_1 then member m_2 to a set has the same result as inserting them in the opposite order [75]. An application which recognizes commutative operations on objects can mechanically handle those messages from the focus handler that specifies updates which commute with the updates made by the application.

Unfortunately, commutativity among pairs of operations is not common except in financial transactions such as "credit" and "debit". So if an application knows

how to handle messages only in such situations, there may be many messages it cannot handle and thus many situations in which the application will be forced to abort operation. Nonetheless, handling even a few messages results in a medium level of cooperation better than the lowest level described above; the more messages an application can handle, the less frequently it will be forced to abort operation and discard updates made by the user.

7.3.3 High level of cooperation

If an application keeps its focus current, and handles all messages received from the focus handler, then it does become possible for that application to operate concurrently with other applications sharing the same database objects and never need to abort because it does not understand an update made by another application. This is a high level of cooperation of the application with the AOM.

An important premise of our work is that forced by the need for a higher degree of concurrency, applications will evolve toward a high level of cooperation. This level of cooperation is difficult to achieve, since an application is not guaranteed that any data which it has read are static; they may be changed at any time by another application operating on behalf of the same or a different user.

A high level of cooperation requires that the application be robust and that it perform reasonably even in situations where data change unexpectedly (due to asynchronous updates by other applications). The framework guarantees that if an application expresses an interest in an update, and that update occurs, then the application will be notified of that update by an asynchronous message from the focus handler.

There are two major benefits from a high level of cooperation.

1. Applications from multiple vendors can be operated concurrently without understanding each others semantics. Instead, an application needs to understand only the semantics of the view of the artifact which it accesses.

2. The user is not forced into a specific order of application invocation, as in the case when exclusive access to objects is employed. The work process can instead be viewed as an evolution, rather than a series of disconnected activities.

7.3.4 Other levels of cooperation

Many levels of cooperation exist between the low and high ends. It is not necessary for an application to have a high level of cooperation with the AOM in order to gain any benefits; it is simply that *a higher level of cooperation derives greater benefit.*

7.4 Conclusion

We conclude this chapter by stating that a co-application, which is linked with the AOM to form an agent useful in cooperative environments, has features which differ from those of conventional applications. These features help surmount cooperative data sharing problems that cooperating product developers are increasingly encountering. We outline the features of a co-application as follows.

- A co-application does not access data in the database directly, but instead manipulate cached copies of objects through a well defined interface.

- It adjusts its focus to include updates to those objects upon which it is currently basing its internal state.

- It responds to messages describing updates made by other applications by incorporating those updates into its internal state and by making compensating updates where necessary in order to restore consistency. If the application defers handling the messages, then it is obliged to handle them before committing updates from the object cache to the Co-DBMS.
- It uses the work status monitor to stay aware of the work status and makes the work status known to its user.
- It gives the user a means to record collisions in order to identify updates by other users as unacceptable.

Chapter 8

Conclusion and Future Work

A central concern in computer supported cooperative work is coordinated access to shared information. In this dissertation, we investigate concurrency control issues for environments that support cooperative work. As a context for our work, we address product development environments where cooperation among a group of diverse and distributed product developers is highly recommended for enhanced productivity. Our research reveals a diverse set of requirements that cannot be supported using conventional applications and their associated database management systems. To support these requirements, we develop a new framework for cooperative data sharing. Contrary to the conventional approach, the operation of the framework considers as a premise the evolution of the product rather than the steps that lead to the product. Another major difference is the replacement of the assumption that users are unrelated and isolated from one another, which underlies the conventional approach, with the fact that product developers communicate with each other, both informally and through the database, to jointly develop the overall product.

The framework is basically comprised of two components: the cooperative

database management system (or Co-DBMS) and the agent. The Co-DBMS consists of an object-oriented data store and a set of modules termed the database object manager (or DOM). The agent has another set of modules, termed the application object manager (or AOM), that are directly linked to any application accessing objects stored in the Co-DBMS. The framework provides a number of desirable features to support cooperative product development.

In this chapter, we review the features of the framework and summarize the main contributions. We also outline several directions for future work.

8.1 Features of the Framework

The framework is open-ended: neither it, nor the applications which make use of it, need to be changed when a new application is introduced into the product development environment. In addition, the framework provides a host of other features through the Co-DBMS and the agent.

8.1.1 Support provided by the Co-DBMS

Since conventional database management systems are inadequate for use in a cooperative product development environment, additional techniques are needed. The DOM of the Co-DBMS adds capabilities to an object-oriented data store to make it suitable for cooperative data sharing. These capabilities are summarized below.

- **Agent registration:** agents register the commencement and termination of their operation with the DOM using connect and disconnect procedures, respectively.

- **Object check-out and check-in:** agents can check-out base objects for read or for update access, and check-in objects they have checked-out when access is no longer required.

- **Asynchronous update notifications:** the DOM sends asynchronous update notifications to agents; these notifications describe updates made by other agents to base objects which have been checked-out.

- **A dynamic workspace hierarchy:** the DOM offers a dynamic workspace hierarchy into which updates may be encapsulated.

- **Support for user mediated consistency:** the DOM enforces constraint specifications that could be modified by the product developers or their agents; the DOM also gives product developers and their agents the ability to mark updates by other agents as collisions, and ensure that a workspace cannot commit to its superior workspace if it contains unresolved collisions.

8.1.2 Support provided by agents

The AOM of an agent provides capabilities to an application which simplify the development of applications that can operate effectively in a cooperative product development environment. These capabilities are summarized below.

- **Consistency of the object cache used by the application**: the AOM keeps the object cache consistent in the face of both internal updates made by the application and external updates performed by other agents.

- **Automatic invalidation of externally derived slots:** the out-date propagator in the AOM automatically invalidates derived external slots when slots upon which they depend are modified; this may cause, for example, constraints throughout a product development hierarchy to be marked as

invalid when some low-level component is modified; thus applications need not be aware of all constraints and other derived slots in the system, nor of the manner in which those slots depend on the values of other slots.

- **Application registers focus and receives messages:** the AOM will monitor changes to objects which an application has included in its focus, and will inform the application when such an update occurs; the application may use this information to make its internal data structures consistent with the object cache.

- **Deferred handling of updates:** a product developer may wish to ignore updates made by agents other than the one he or she is using; this can be done because the AOM gives the application a programmatic interface to defer the incorporation of external updates into the object cache.

8.2 Research contributions

This section summarizes the research contributions of this dissertation.

8.2.1 Object model for cooperative product development databases

We developed an object model and associated operations on objects which can be used as a basis for a more complete object-oriented database. The object model described in this dissertation is a formal model, and was used, throughout the dissertation, to explain the operations of the framework. The model considers objects as being inter-related and attaches additional information to objects in order to facilitate cooperative work.

- **Relationships among objects**: in addition to providing support for nested-objects, object references, and set-valued objects, the model provides derived objects to represent the semantics of inter-object relationships.

- **Control attributes**: these are ancillary information attached to objects in order to represent the difference between the state of objects in a workspace and the state of those objects in its superior workspace; the Co-DBMS uses this information to compute the update delta when a workspace is to be committed to its superior workspace; the same control attributes enable an agent to track how objects in its cache differ from those in the database; the agent uses this information to compute the update delta when it needs to commit its updates to the Co-DBMS.

8.2.2 Flexible model of concurrency control

The major contribution of this research is the development of a flexible model of concurrency control, which does not necessitate the use of exclusive access; the absence of exclusive access makes a high degree of cooperation possible among a group of product developers who are collaboratively completing a product. The main features of the model are outlined below.

- **Use of notification**: the update monitor in the DOM provides the mechanism through which an agent can become aware of updates made by other agents to cached objects. The DOM sends asynchronous update notifications to the agents which describe updates that occur.

- **Applications handle Notifications**: when applications use this mechanism and follow the requirements of a co-application, explained in Chapter 7, they can keep their internal data structures consistent with the state of the objects in the Co-DBMS without the need for restrictive exclusive access;

the update notification manager in the AOM automatically incorporates updates from other agents into the object cache, but the application has the responsibility of updating its internal data structure.

- **Multiple levels of cooperation:** applications can exhibit varying degrees of cooperation; a range of techniques is possible, all of which guarantee consistency; levels of cooperation differ in the amount of application-specific knowledge required; use of knowledge offers a high level of cooperation and enables an application to respond flexibly to update notifications rather than abort operation.

8.3 Future Work

Our research work unveils a number of important areas for future work in information sharing in CSCW generally and our framework specifically. In this section, we outline some of these areas.

8.3.1 Prototype of the framework

Our proposed framework, and its associated mechanisms, represent a new approach to achieve cooperative data sharing. The construction of a prototype framework and co-applications is important to establish a proof of concept for feasibility and effectiveness of our approach to cooperative product development environments.

Some concepts that the prototype will demonstrate are:

- automatic handling of update notifications to an agent's object cache by the update notification manager, the out-date propagator, and the derived slot calculator in the agent;

- changes are propagated among applications which share updates to the same objects;
- the application maintains the consistency of its internal state in face of concurrent updates.

8.3.2 Application of domain-specific semantics

The semantics of a particular domain, for example, software development, can be used to develop views, interests, constraints, and methods which employ knowledge of that domain. Modules which offer domain-specific capabilities could, like the AOM, be included with each application and would simplify the task of the application developer.

Abstraction is one way of exploiting domain-specific knowledge. It is possible, for example, to apply abstraction to interests. An interest abstraction is a high level interest which is translated from a domain-specific level to a set of more primitive interests. Using interest abstractions, a programmer who develops applications will have more powerful vocabulary with which to express updates on products, and therefore reduce the complexity which the programmer must handle.

Domain-specific semantics could also be used by the focus handler for the efficient handling of the message queue. An extension to the focus handler could be to enable it to recognize messages which represent sets of operations that could be performed using a less number of operations. For example, mechanisms could be added to the focus handler to recognize inverse and idempotent operations and prune the queue appropriately. Efficient handling of the message queue evidently reduces the number of messages which the application has to handle [23].

8.3.3 A framework for handling shared messages

Message passing (not to be confused with messages passed from the AOM to the application) is yet another important way for information sharing. Several research and development efforts have been geared toward enhancing the capabilities of electronic mail to better suit cooperative work [51, 24]. The concepts provided in this dissertation could also be used to achieve this goal.

While we developed the framework for cooperative sharing of database objects, the architecture of the framework could also provide an infrastructure for public and directed message handling. This can be done by having a message handler, similar to the Co-DBMS, receive messages from agents acting on behalf of users and then handle these messages accordingly; if the message is public, then it could be read by any other agent; if the message is directed, however, the message handler will re-direct the message to the particular agent(s) to whom the message is addressed. An additional twist could be to have agents register their interest in specific messages (for example, a specific subject), when the message handler receives messages on a subject that matches an interest, it sends these messages to the agent(s) who registered that interest. Messages could also have validity conditions (for example, expiration time). The message handler should ensure that messages read or received by agents satisfy the validity conditions. We are currently conducting an investigation to identify the requirements of message handling in cooperative environments and the framework components that will provide the features needed to support these requirements.

Bibliography

[1] E. Adams, M. Honda, and T. Miller. Object management in a CASE environment. In *11th International Conference on Software Engineering*, pages 154–163. IEEE Computer Society Press, 1989.

[2] D. Agrawal and A. Elabbadi. Transaction management in database systems. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 1–32. Morgan Kaufmann Publishers, San Meteo, CA, 1992.

[3] J. Austin. *How to Do Things with Words.* Harvard Press, 1962.

[4] R. Baecker, editor. *Readings in Groupware and Computer-Supported Cooperative Work.* Morgan-Kaufmann Publishers, San Meteo, CA, 1993.

[5] F. Bancilhon, W. Kim, and H. Korth. On long duration CAD transactions. In D. Maier and S. Zdonik, editors, *Readings in Object-Oriented Database Systems*, pages 408–431. Morgan Kaufman Publishers, San Meteo, CA, 1990.

[6] N. Barghouti and G. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 33(3):269–317, September 1991.

[7] P. Bernstein, A. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[8] A. Bond. Cooperation in artifact design. In *MIT-JSME Workshop*, 1989.

[9] H. Bonin. Teamwork between non-equals: Check-in and check-out model. *ACM SIGOIS Bulletin*, 13(3):18–27, December 1992.

[10] A. Buchmann and C. de Celis. An architecture and data model for CAD databases. In *Proceedings of VLDB 85*, pages 105–114, 1985.

[11] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill Book Company, 1984.

[12] P. Dewan and J. Riedl. Toward computer-supported concurrent software engineering. *IEEE Computer*, 26(1):12–15, January 1993.

[13] E. Dyson. A framework for groupware. In D. Coleman, editor, *Groupware'92.* Morgan Kaufmann Publishers, San Meteo, CA, 1992.

[14] C. Ellis and S. Gibbs. Concurrency control in groupware systems. In *ACM SIGMOD'89 Conference on the Management of Data*, 1989.

[15] C. Ellis, S. Gibbs, and G. Rein. Design and use of a group editor. In *Engineering for Human-Computer Interaction*, pages 13–25. North Holland, Amsterdam, 1990.

[16] C. Ellis, S. Gibbs, and G. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1990.

[17] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufman Publishers, San Meteo, CA, 1992.

[18] A. Elmagarmid and C. Pu. Introduction to the special issue on heterogeneous databases. *ACM Computing Surveys*, 22(3):175–178, September 1990.

[19] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–632, November 1976.

[20] M. Fernandez and S. Zdonik. Transaction groups: A model for controlling cooperative transactions. In *Proceedings of the 3rd International Workshop on Persistent Object Systems*, Berlin, Germany, 1989. Springer-Verlag.

[21] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2), June 1983.

[22] H. Garcia-Molina and K. Salem. SAGAS. In *ACM SIGMOD International Conference on Management of Data*, pages 249–259, 1987.

[23] G. Ghung, K. Jeffay, and H. Abdel-Wahab. Accommodating latecomers in shared window systems. *IEEE Computer*, 26(1), January 1993.

[24] Y. Goldberg, S. Marilyn, and E. Shapiro. Active mail: A framework for implementing groupware. In *ACM 1992 Conference on Computer-Supported Cooperative Work*, pages 75–83, 1992.

[25] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufman Publishers, San Meteo, CA, 1993.

[26] I. Greif, editor. *Computer-Supported Cooperative Work: A Book of Readings.* Morgan Kaufman Publishers, San Meteo, CA, 1988.

[27] I. Greif. *Designing Group-Enabled Applications: A Spreadsheet Example*, pages 515–525. Morgan Kaufman Publishers, San Meteo, CA, 1992.

[28] I. Greif and S. Sarin. Data sharing in group work. In Irene Grief, editor, *Computer-Supported Cooperative Work: A Book of Readings*, pages 477–508. Morgan Kaufman Publishers, San Meteo, CA, 1988.

[29] J. Grudin. CSCW: the convergence of two development paradigms. In *ACM CHI'91*, pages 91–97, 1991.

[30] J. Grudin. The CSCW forum. In *26th Annual Hawaii International Conference on System Sciences*, pages 51–58, 1993.

[31] W. Harrison, H. Ossher, and P. Sweeney. Coordinating concurrent development. In *ACM 1990 Conference on Computer-Supported Cooperative Work*, pages 157–168. ACM SIGCHI and SIGOIS, 1990.

[32] S. Heiler, S. Haradhvala, S. Zdonik, B. Blaustein, and A. Rosenthal. A flexible framework for transaction management in engineering environments. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 87–122. Morgan Kaufman Publishers, San Meteo, CA, 1992.

[33] H. Jagadish and O. Shmueli. A proclamation-based model for cooperating transactions. In Li-Yan Yuan, editor, *18th International Conference on Very Large Databases*, pages 265–276, 1992.

[34] R. Johansen. *Computer Support for Business Teams.* The Free Press, 1988.

[35] R. Johansen. *Leading Business Teams.* Addison-Wesley, Reading, MA, 1991.

[36] P. Johnson-Lenz and T. Johnson-Lenz. Groupware: The process and impact of design choices. In Kerr and Hiltz, editors, *Studies of Computer-Mediated Communication Systems*, pages 45–55. 1982.

[37] G. Kaiser. A flexible transaction model for software engineering. In *IEEE International Conference on Data Engineering*, pages 560–567, 1990.

[38] G. Kaiser and D. Perry. Workspaces and experimental databases: Automated support for software maintenance and evolution. In *Conference on Software Maintenance*, pages 108–114. IEEE Computer Society Press, 1987.

[39] W. Kim and F. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications*. ACM Frontier Series. Addison-Wesley Publishing Company, Reading, MA, 1989.

[40] H. Korth, F. Levy, and A. Silberschatz. Compensating transactions: A new recovery paradigm. In *16th VLDB Conference*, pages 95–106, 1990.

[41] H. Korth and G. Speegle. Long duration transactions in software design projects. In *IEEE International Conference on Data Engineering*, pages 568–574, 1990.

[42] C. Krueger. Persistent long-term transactions for software development. CMU-CS-90-188, Carnegie Mellon University, November 1990.

[43] A. Kumar and M. Stonebraker. Semantic-based transaction management techniques for replicated data. In *ACM SIGMOD International Conference on Management of Data*, pages 117–125, June 1988.

[44] M. Kumar and J. Wong. Concurrency control in design databases. TR#91-05, Iowa State University, February 1991.

[45] H. Kung and J Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–236, June 1981.

[46] C. Kydd and D. Ferry. A behavioral view of computer supported cooperative work tools. *Management Systems*, 3(1):55–67, 1991.

[47] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, July 1978.

[48] F. Londono. *A Blackboard Framework to Support Concurrent Engineering.* PhD thesis, West Virginia University, 1990.

[49] D. Maier. Making database systems fast enough for CAD. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 573–592. Addison-Wesley Publishing Company, 1989.

[50] D. Maier and S. Zdonik, editors. *Readings in Object-Oriented Database Systems.* Morgan Kaufman Publishers, San Meteo, CA, 1990.

[51] T. Malone, K. Grant, F. Turbak, S. Brobst, and M. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, May 1987.

[52] J. Manzi. *Working Together*, pages 3–9. Morgan Kaufman Publishers, San Meteo, CA, 1992.

[53] J. McGrath. *Groups: Interaction and Performance.* Prentice Hall, Englewood Cliffs, NJ, 1984.

[54] E. Moss. *Semantics for Transactions in Shared Object Worlds*, pages 289–294. ACM Frontier Series. Addison-Wesley Publishing Company, Reading, MA, 1990.

[55] S. Mujica. *A Computer-Based Environment for Collaborative Design.* PhD thesis, University of California, Los Angeles, 1991.

[56] K. Narayanaswamy and N. Goldman. Lazy consistency: A basis for cooperative software development. In *ACM 1992 Conference on Computer-Supported Cooperative Work*, pages 257–264, 1992.

[57] M. Nodine, S. Ramaswamy, and S. Zdonik. A cooperative transaction model for design databases. In A. Elmagarmid, editor, *Database Transaction Models*

for Advanced Applications, pages 53–86. Morgan Kaufman Publishers, San Meteo, CA, 1992.

[58] M. Nodine and S. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *VLDB*, pages 84–90, 1990.

[59] J. Nunamaker, A Dennis, J. Valacich, D. Vogel, and J. George. Electronic meeting systems to support group work. *Communications of the ACM*, 34(7):40–61, July 1991.

[60] L. Osterweil. Software environment research: Directions for the next five years. *IEEE Computer*, 14(4):35–43, 1981.

[61] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MD, 1986.

[62] C. Pu, G. Kaiser, and N. Hutchinson. Split transactions for open-ended activities. In *14th International Conference on VLDB*, pages 26–37. Morgan Kaufmann Publishers, 1988.

[63] R. Reddy, K. Srinivas, V. Jaganathan, and R. Karinthi. Computer support for concurrent engineering. *IEEE Computer*, 26(1):12–15, January 1993.

[64] R. Reddy and R. Wood. Emerging prototypes for concurrent engineering. In *Second National Symposium on Concurrent Engineering*, 1990.

[65] B. Reeves and F. Shipman. Supporting communication between designers with artifact-centered evolving information spaces. In J. Turner and R. Kraut, editors, *ACM 1992 Conference on Computer-Supported Cooperative Work*, pages 394–401, 1992.

[66] T. Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[67] J. Rosenberg and D. Koch, editors. *Persistent Object Systems.* Springer-Verlag, 1989.

[68] A. Skarra. *A Model of Concurrency Control For Cooperative Transactions.* PhD thesis, Brown University, 1991.

[69] P. Sorgaad. A framework for computer supported cooperative work. In J. Kaasboll, editor, *Report of the 11th IRIS Seminar*, pages 620–640. University of Oslo, 1988.

[70] S. Sutton. A flexible consistency model for persistent data in software-process programming languages. In *4th International Workshop on Persistent Object Systems.* ACM Press, 1990.

[71] A. Tanenbaum. *Computer Networks.* Prentice-Hall Inc., Englewood Cliffs, NJ, second edition, 1988.

[72] R. Taylor, L. Clarke, L. Osterweil, J. Wileded, and M. Young. Arcadia: A software development environment research project. In *IEEE 1986 ADA Applications and Environments Conference*, 1986.

[73] D. Tylor. *Object-Oriented Information Systems: Planning and Implementation.* John Wiley and Sons, Inc., New York, NY, 1992.

[74] R. Unland and G. Schlageter. A transaction manager development facility for non standard database systems. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 399–466. Morgan Kaufmann Publishers, San Meteo, CA, 1992.

[75] W. Weihl. Commutativity-based concurrency control for abstract data types. In *21st Annual Hawaii International Conference on System Sciences*, pages 205–214. IEEE Computer Society Press, 1988.

[76] R. Winner, J. Pennel, H. Bertrend, and M. Slusarczuk. The role of concurrent engineering in weapons system acquisition. IDA Report R-338 (DTIC#AD-A203-615), IDA, Alexandria, VA, 1988.

[77] T. Winograd. A language/action perspective on the design of cooperative work. *Human-Computer Interactions*, 3(1):3–30, 1987.