CIFE CENTER FOR INTEGRATED FACILITY ENGINEERING

A FRAMEWORK FOR CHANGE MANAGEMENT IN A DESIGN DATABASE

by

Keith Hall, Gio Wiederhold and Kincho Law

TECHNICAL REPORT
Number 65

August, 1991

Stanford University

A FRAMEWORK FOR CHANGE MANAGEMENT IN A DESIGN DATABASE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By

Keith Hall

August 1991

Abstract

The complexity of many engineering designs is now so great that it is impossible for one engineer to complete a design alone. Instead, a team of engineers work collaboratively to achieve the design. This is known as *concurrent design*. Despite evolution of computer-aided design (CAD) tools toward integrated use of a design database, tools continue to assume that they will operate in isolation from one another and will take turns accessing and manipulating design data. The result is a *tool-centric approach* that makes close cooperation among a team of design engineers impossible.

This thesis argues that a tool-centric approach is unsuitable in a design environment for concurrent design, which requires a new *design-centric approach* that does permit close cooperation, and that existing tools and design databases cannot offer that new approach. As a framework for the design-centric approach, this thesis presents the Change Manager, which is a collection of software modules that are distributed among CAD tools and the design database. The Change Manager augments the functionality of tools and the database by serving as an intermediary between application code in a tool, and design data in the database.

The Change Manager offers a flexible model of concurrency control without the need for exclusive locks, and also provides support for close cooperation of engineers and their tools. The collaboration can be informal—directly among the engineers—as well as formal—computer-mediated. The Change Manager provides mechanism and protocol, but does not enforce policy. This thesis describes the architecture and operation of the Change Manager, explains how it supports a design-centric approach, and presents invariants that its operation guarantees.

Acknowledgements

The experience of getting a Ph.D., that is, surviving coursework, locating a research topic, exercising gray matter, writing a small book, taking exams, and jumping through various flaming hoops, is an arduous one. Many people have provided invaluable technical assistance or emotional support. In this section I thank a few of these people. I also greatly appreciate financial support from National Science Foundation under grant DMC-8619595 "Support for Parallel Design in an Engineering Information System".

Gio Wiederhold, my principal advisor, made sure that my research efforts were of technical merit. He introduced me to interesting research areas and identified dead ends. Gio also frequently adjusted my bearings away from system administration and toward the acquisition of a doctoral degree. Thesis reader Kincho Law braved many incomplete drafts of this thesis and greatly improved its exposition. Kincho also repeatedly dislodged my perspective from the ether and made this research practical. Tore Risch, another thesis reader, enthusiastically discussed and critiqued new ideas. Tore's constant good humor enabled my optimism toward eventual parole from graduate school. My officemate Peter Rathmann generously offered and patiently provided his clear thinking to many of my ideas. Peter is also responsible in part for my addiction to Thai iced coffee.

Vic, Sheri, Lisa, Brian, and Patty Hall supplied confidence in my ability to finish school within this millenium. Voy Wiederhold provided positive energy that made steady progress possible, and gave me furniture to move when I needed exercise. Bill McKinnon located jobs for me when money was tight, gave me refuge when I needed to escape from research, and taught me hand signals for refereeing soccer games. Ellen Doyle encouraged me to persevere at times when I had given up, and extracted me from snow after skiing mishaps.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The computer-based design environment to support engineering tasks, which consists of computer-aided design applications, storage of design data, and an associated methodology, is evolving from the use of numerous data files to the use of a central database. Even with this evolution, emphasis continues to be placed on providing support for tools to operate in isolation from one another and to take turns accessing and manipulating design data; this is a **tool-centric** approach.

This thesis argues that a tool-centric approach is unsuitable for a design environment for concurrent design, which requires a **design-centric** approach. In a design-centric approach, engineers can use multiple tools to effect changes to the same design or portion of a design simultaneously. Design centricity places special requirements on computer-aided design (CAD) tools and the underlying database for design data. Existing tools and design databases cannot offer a design-centric approach.

As a framework for the design-centric approach, this thesis presents the **Change Manager**. The Change Manager is a revolutionary approach which uses asynchronous notifications, rather than exclusive locking, in order to maintain consistency among tools. The Change Manager is implemented as a collection of software modules which are distributed among computer-aided design (CAD) tools and the data storage system for design data. The Change Manager enables application code to perform as the "CAD tool of the future", and

converts the data store to the design database of the future—the **Design Data Management System** or DDMS.

Note that a database to support a design-centric approach need not be physically centralized. There is extensive literature on the management of physically distributed data, and we need not consider distributed databases [Ceri 84] here.

The functionality required of CAD tools that would support concurrent design were investigated in order to identify what services should be provided by the Change Manager. The Change Manager adds capability both to CAD tools and to the DDMS. It provides mechanism and protocol, but does not enforce policy. It serves as an intermediary between application code in a tool, and design data in the data store.

The architecture and semantics of the Change Manager are independent of the semantics of particular tools. Furthermore, the Change Manager does not require that tools understand the semantics of other tools; each tool need understand only the view of design data that it manipulates [Stefik 82]. The Change Manager does not assume a particular engineering domain, such as VLSI or software. It does assume an overall model of cooperating activities during the design process, as exercised by participating engineers; that model of interaction is presented in this chapter.

Part 1.A: Overview of Thesis

This thesis consists of eight chapters. Chapter 1 introduces fundamental concepts such as concurrent design, design data management system (DDMS), and CAD tool, and concludes with an overview of related research. Chapter 2 defines the object model used to represent design data within the DDMS. The object model provides a formal basis for later chapters.

Chapter 3 motivates research in concurrent design by identifying ways in which existing database systems and CAD tools are inadequate to provide the design-centric approach re-

quired for a concurrent design environment, and offers justification of design decisions of the high-level architecture of the Change Manager.

Chapter 4 describes the architecture of the DDMS, explains what capability the Change Manager adds to the data store in order to produce the Design Data Management System, specifies a programmatic interface to that capability, and describes invariants maintained in the DDMS by the Change Manager. Chapter 5 describes the architecture of a CAD tool, explains what capability the Change Manager adds to an application in order to produce a CAD tool that interfaces to the DDMS, specifies a programmatic interface to that capability, and describes invariants maintained in tools by the Change Manager.

Chapter 6 defines what is minimally required of an application in order to be well-behaved, and describes various levels of coordination of applications with the Change Manager.

Chapter 7 summarizes key ideas of the preceding chapters, identifies research contributions, and describes ongoing and future work on the Change Manager.

Part 1.B: Concurrent Design

Section 1.B.1: Complexity of Designs Requires Concurrent Design

Because of the complexity of engineered artifacts, the associated design phase is a large effort. Design activities are shared among a team of **engineers** whose work is coordinated by a **supervisor**. Each engineer works under and reports to a supervisor. The same person can assume both roles, but we keep them distinct.

In order to accomplish design tasks, engineers employ **CAD tools** to manipulate **design data**. The complexity of engineering designs has rapidly increased, and will continue to increase. Many engineering designs are now so complex that it is impossible for one engineer, using one or two CAD tools, to complete the design alone. Instead, a team of

engineers work together and use a suite of tools. See Figure 1. Members of this team work in parallel on different but related aspects of the design. This is known as **concurrent design**.

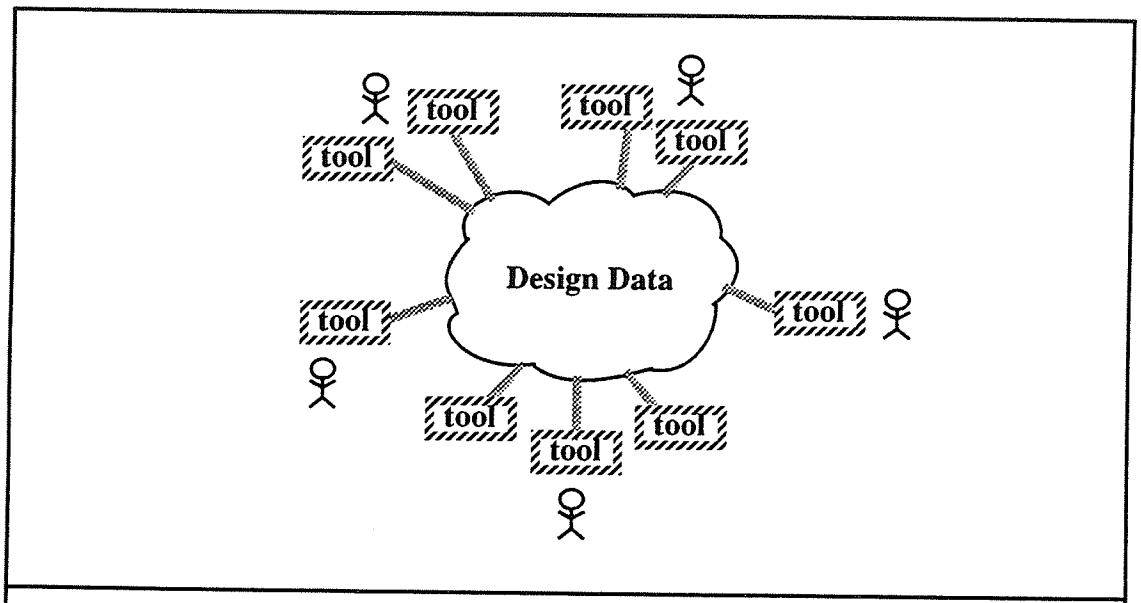

Figure 1: Concurrent design.
A team of engineers work in parallel on different aspects of the same design. CAD tools provide the means for the engineers to manipulate the design data which represent the design.

An important aspect of concurrent design is that these parallel activities are *cooperative*, that is, there exist *communication* and *cooperation* among design engineers in order to achieve a common higher goal. That higher goal is the parent design, which is a combination of the work done by individual members of the design team. This approach contrasts sharply with transaction models commonly used in databases [Ullman 82].

Section 1.B.2: Conflicting Updates

In concurrent design, engineers release their progress on the design to other members of the design team at intervals rather than continuously, since an engineer's state of progress is often incomplete and experimental. Furthermore, engineers cannot be always fully aware

of the impact of changes they make on the overall consistency of the design; aspects of consistency are modeled by **constraints** on the design. As a result, the efforts of one engineer may conflict with those of another. If the engineers cannot resolve the conflict themselves, their supervisor must resolve the conflict.

In existing CAD tools, constraint checking is performed in a haphazard fashion, and engineers and the tools they run are not provided with systematic assistance to become aware of changes to the validity of constraints. Nor are engineers made aware of the occurrence of other events, such as the creation of new versions of a subcomponent.

Section 1.B.3: Design Partitioning

It is the supervisor's job to **partition** a design into smaller portions, each less complex and of more manageable size. These portions of the original design are called **components** of the parent design. Each component is an instance of another design called a **subdesign** of the parent design. A component references the associated subdesign and provides parameters, such as displacement and rotation, which characterize that particular instance of the design within its containing design. Partitioning can be applied recursively to subdesigns, resulting in a hierarchy of components or a directed acyclic graph of subdesigns. After performing partitioning, the supervisor distributes work on the subdesigns among the engineers, and assigns permissions to access and update designs. See Figure 2.

A well-partitioned design offers several benefits:

- After the functionality needed in a subdesign has been identified, the process of design can proceed in parallel on the subdesigns and on the parent design. The completion of a subdesign is a task smaller than that of completing the entire design before it was partitioned.
- Identifying subdesigns whose functionality is generally useful means that multiple instances of these subdesigns can be used in the parent design or in other design projects.

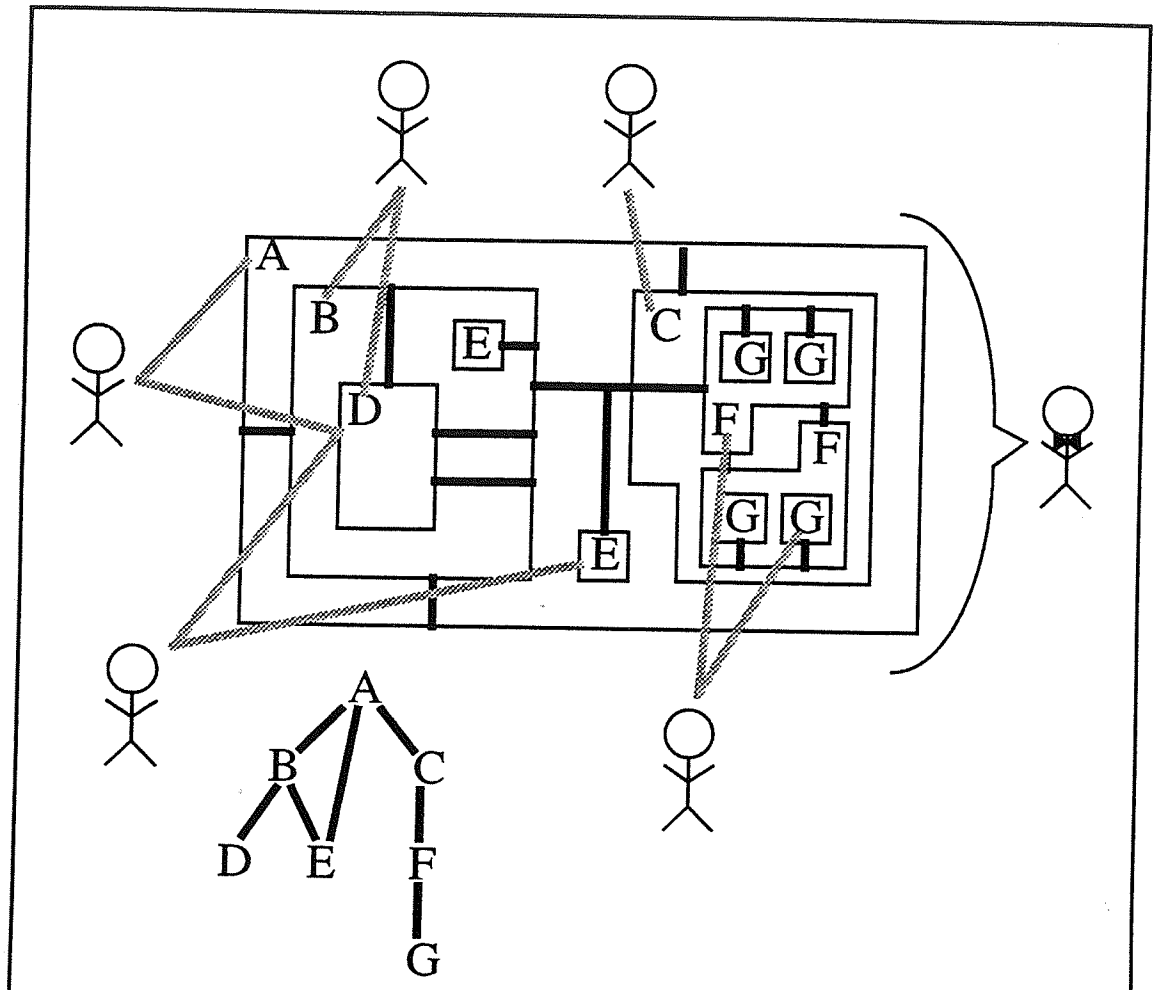

Figure 2: Design partitioning.
The supervisor divides design A into a hierarchy of instances of subdesigns B–G, which are themselves designs with their own components (and associated subdesigns). Work on subdesigns proceeds in parallel.

- Subdesigns are themselves designs which can have their own subdesigns. Identifying subdesigns reduces the complexity with which engineers must cope at any one time; an engineer need ensure only that the (sub)design being completed meets its specifications and can assume that all of its subdesigns will meet their specifications.
- The supervisor can use the 'design' (or subdesign) as the unit of protection. That is, the supervisor can assign access and update permissions to specific engineers for specific (sub)designs.

Components in a partitioned design are interrelated, since they jointly implement the functionality of the parent design. To the extent possible, however, the partition is chosen so that interactions and interdependencies among subdesigns is minimized; this is done so that the amount of communication required among the engineers working on the subdesigns, as well as the likelihood of conflict, is reduced.

Part 1.C: Evolution of the Design Environment

The environment in which engineers work and the protocol they use in order to achieve their designs is called the **design environment**. This part of the chapter will describe the evolution of the design environment in recent years from the use of data-files to the use of a design database.

Section 1.C.1: Data-Files

Most CAD tools in use today were developed by different vendors with different goals, and before the importance of interoperability was recognized. For this reason, emphasis was placed on the functionality of that one tool, that is, on the manipulations of design data which that tool would perform. The fact that other tools might need to perform manipulations on related aspects of the design, or even that other tools exist, was not initially considered.

The tools which resulted from this insular philosophy have their own private repositories of design data [Mehmood 87]. These repositories are collections of files. The files are *opaque*, that is, their semantic content is unknown to all but the one tool which uses those files and for which the file formats were developed. Thus interrelationships among the tools' data sets, which may represent multiple aspects of the same design, are ignored and it is impossible to automatically maintain consistency among these views. Instead, correspondence among various files must be manually maintained. Doing so in a setting of con-

current design, that is, involving a number of engineers, is a complex, time-consuming, and error-prone task [Wiederhold 86b][Wiederhold 88].

Many vendors have now made public the formats of files used by their CAD tools. This openness has motivated the creation of a new market, that of CAD tool "integration", in which translation utilities or "filters" are developed to convert from one vendor's file formats to another's. See Figure 3.

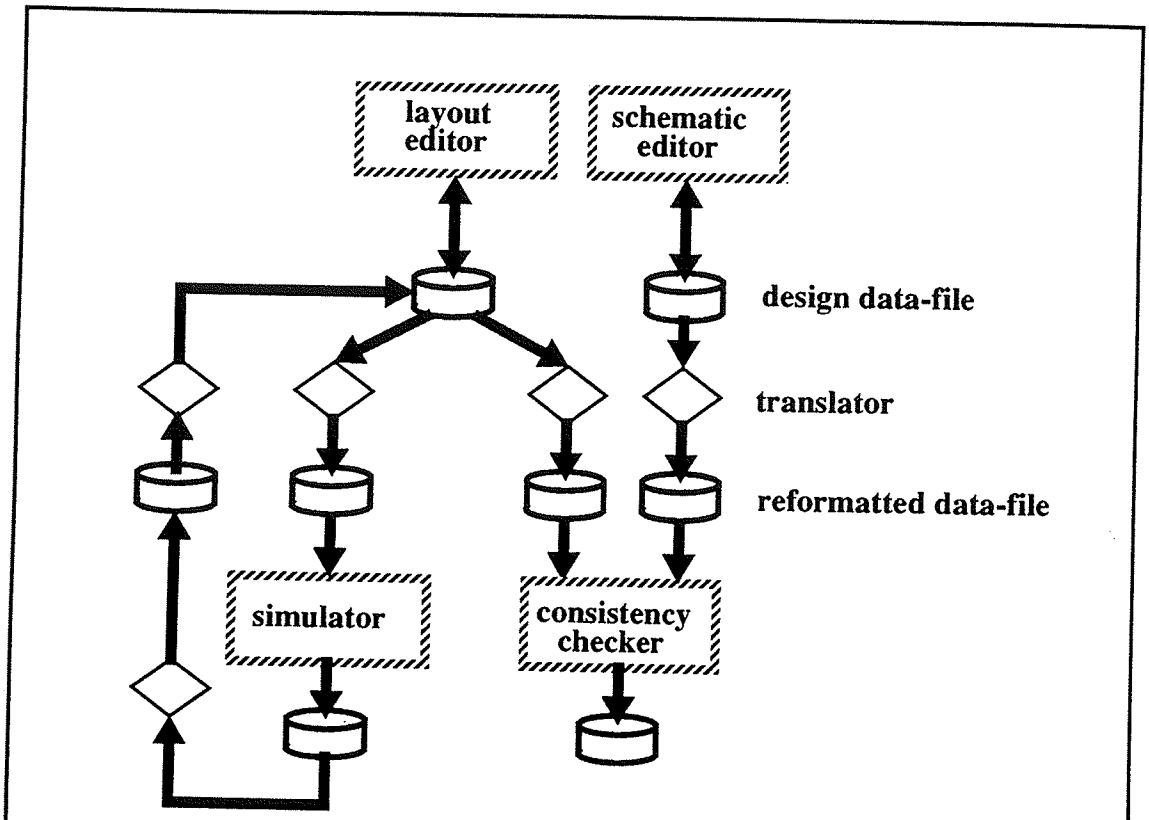

Figure 3: CAD tool file formats.
Each tool requires a particular format for its data, and filters are developed in order to translate among some of the formats.

Absence of communication among tool vendors during development of tools 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 these filters, various standards committees are actively defining standard file formats which implement common views of designs. The electronic design domain, for example, has standards such as EDIF and VHDL. CAD tools which use standard file formats along with the filters to translate among the various formats are collectively referred to as a "CAD framework". The advantage of this approach is that since the CAD tools do not require modifications to use the framework, companies' investments in existing tool suites are preserved.

A CAD framework, with its ability to translate among common file formats, is an important first step toward interoperability of CAD tools and thus the sharing of design efforts among a team of engineers. But such a framework fails to offer a single common integrated data model of design data that all tools can use.

In some cases one view of a design may be automatically derived from another. For example, some CAD tools can synthesize an integrated-circuit layout from a schematic. The comparison of the timestamp of a derived file to the timestamp of the source file(s) permits recognition that a derived file is out-of-date. Note, however, that this permits re-derivation only at a very coarse granularity, namely at the level of an entire file. This is combined with the fact that tools typically operate on their data by reading and writing entire files rather than by making incremental updates. Limiting the granule size to entire files eliminates support for performing incremental analysis on evolving designs. Change notices to interested parties (such as team members or supervisors) are also restricted to the coarse level of detail "file *X* has changed". If a tool does not automatically recompute derived files then the design engineer must repeat manual procedures previously performed.

The large granularity of change in the data-file approach is a serious weakness of that approach. Furthermore, since concurrent updates to different parts of the same file by two or more engineers will result in inconsistencies, and the unit of design data is the "file", two

design activities within a framework can proceed concurrently only to the extent that they involve different, unrelated files.

Section 1.C.2: The Database Approach

In the data-file approach to design, emphasis was placed on the use of a particular tool at a particular time and on translating design data into a format suitable for that tool. The usefulness of these tools is thus limited, because the design data they manipulate are not integrated. The data management needs of the design environment are extensive and complex. The need in the engineering design environment for capability which traditionally has been associated with a database management system, such as structured information, an integrated data model, concurrency control, transactions, and controlled access to data, has become apparent in the last few years [Brodie 84][Katz 83].

Meta-Data

Design data include both engineering data which represent physical quantities within a design, and meta-data which arise from relationships among engineering data and from design status [Batory 85][Wiederhold 80b]. Examples of meta-data are:

Relationships: There are dependencies and constraints among engineering data which must be stored and manipulated. How design data are related affects how a team of engineers coordinate their activities. The following are examples of relationships among design data:

- structural relationships: The use of a design as a component in another design (the *part-of* relationship).
- functional dependencies: The bounding box of a layout can be computed from the bounding boxes of its components and the size and location of their interconnections.
- constraints: The equivalence between the layout and schematic views of a design.

Design status: The supervisor sees design as a cooperative effort among several engineers. However, even in a spirit of cooperation it is commonplace for one engineer to modify data which are used by another (e.g., one designer changing the implementation of a component). Thus, knowledge that an engineer's tool is updating a design can be useful to other engineers. This is an example of **design status**, and is generated by use of the check-out/check-in protocol with which tools specify an intent to update a design. Another example of design status is a complaint made by a tool, possibly initiated by the engineer using that tool, that another tool has made a change which has repercussions detrimental to progress on the complaining tool's part of the design; this is called a **conflict**. Which conflicts are outstanding is another example of design status and is also useful knowledge for engineers and their supervisors.

Uniform Data Model

Placing design data in a database makes them available for use by many tools and engineers. The database provides the same programmatic interface and integrated data model to all tools [van der Meijs 85]. Tools read and update the data in the database, and during their operation cache their own views of those data; such a view enables the tool to efficiently perform its task. Each tool derives the view it needs from the integrated data model offered by the database. Conversely, when a tool needs to effect 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 tool. A filter is tool-specific and is developed by the tool vendor rather than by a tool integrator, and is thus part of the tool. See Figure 4.

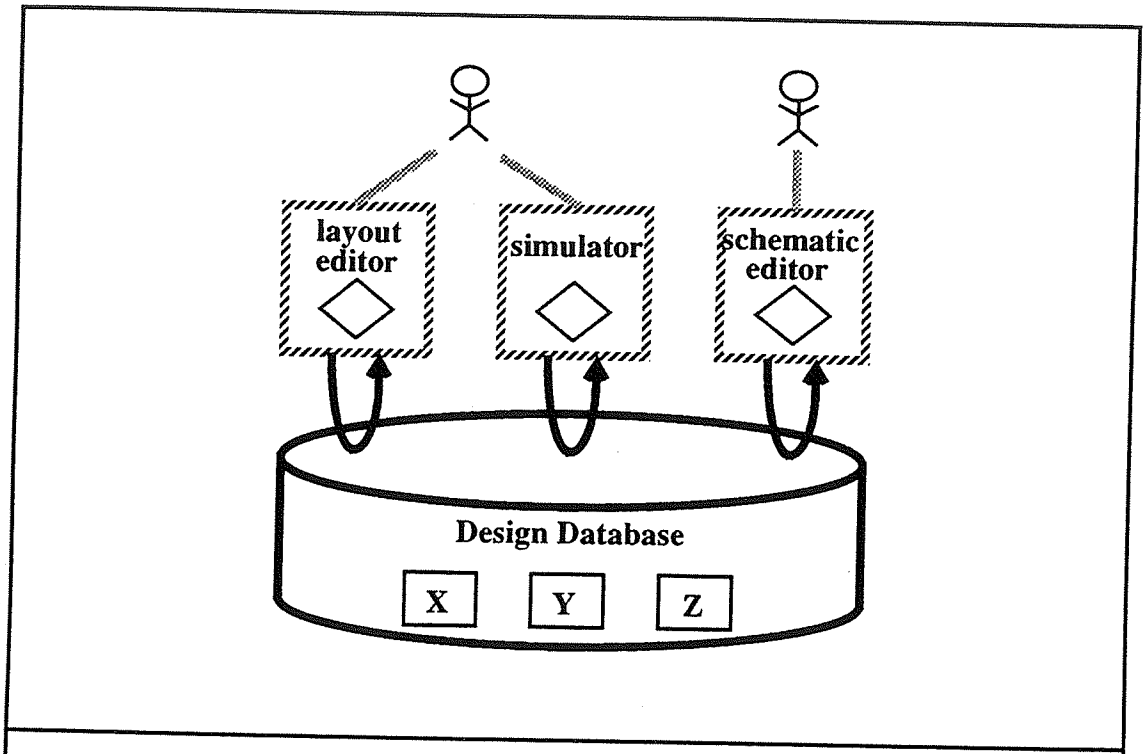

Figure 4: The database approach to design environments.
CAD tools employ the same programmatic interface to access and manipulate design data. Translators internal to the tools convert from the common data model to special-purpose data structures as needed by the tools.

Advantages Offered by Database Technology

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

- The integrated data model of the database is advertised. Any tool vendor is free to develop tools which adhere to that model and which manipulate design data.
- A database accepts incremental updates. Thus a tool that updates a portion of a design need not re-enter the entire design. Instead it can submit only those updates which represent the delta of change effected by the tool on the design. This aspect of databases is particularly important to the utility of the Change Manager, as will be explained in later chapters.

- The database serves as central point where a supervisor specifies which engineers have update access to designs.
- A database offers stable storage of design data, and typically includes techniques to ensure high availability of data in the event of hardware failures, and the ability to rollback to previous states or undo recent changes.

Limitation of Traditional Database Systems

In a traditional database, great care is taken to ensure 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* [Papadimitriou 86]. In the design environment this would mean that concurrently executing tools could not effect progress on the same design. Furthermore, existing tools operate under the assumption that they have exclusive access to the data they manipulate. But *these characteristics are counter to the premise of concurrent design*, in which multiple engineers use multiple tools to complete the design as a team. Thus, a traditional database and traditional tools are inadequate in a design environment which supports concurrent design. The next part of this chapter describes a hypothetical design environment which does support concurrent design. Remaining chapters of the thesis constitute an exposition of *how* to create such a design environment.

Part 1.D: Next-Generation Design Environment

As mentioned in the preceding section, in a design environment which makes use of a traditional database and traditional tools, there is an underlying assumption that a tool which accesses design data has exclusive access to those data. But this is quite restrictive: at any given time the tools that an engineer can invoke strongly depend upon which tools are presently in use by that and by other engineers. Said in another way, concurrent operation of tools must be carefully controlled in order to retain consistency of the designs. This is a

tool-centric approach. It should be the evolution of a design, rather than the interdependencies among tools used, that is the focus of an engineer's efforts.

Design activities involve the use of CAD tools to effect progress on the design; these tools may involve any of presentation or synthesis or analysis of design data, and as such provide the interface through which engineers inspect and manipulate the design. In concurrent design, the engineers plan, communicate, and cooperate, and the tools under their control are used collectively to achieve the design. This we call a *design-centric* approach.

The importance of concurrent design has only recently been recognized [Wiederhold 86b]. A design database and CAD tools that support concurrent design must permit operations of arbitrary length which do not preclude access to design data by other engineers. For this reason, and for additional reasons to be presented in Chapter 3, traditional techniques for databases and tools fail to adequately support concurrent design. Concurrent design requires a new design environment. This "next-generation" design environment should facilitate a collection of design activities proceeding concurrently under the control of collaborative team of engineers.

This part of the chapter elaborates the model of the next generation of design environment, heretofore called simply "design environment", by defining entities involved and describing their relationships. That model is the setting upon which architectural decisions presented in the following chapters are based. The model is specific neither to particular CAD tools nor to a particular engineering domain such as VLSI design or software, in order that it be generally applicable. At the highest level, the design environment consists of **agents** using instances of **tools** to access the **design data management system** (or **DDMS**). Each of these elements is discussed below. See Figure 5. This figure and other figures in this chapter use structural relationships to model relationships among entities in the design environment [Wiederhold 80a].

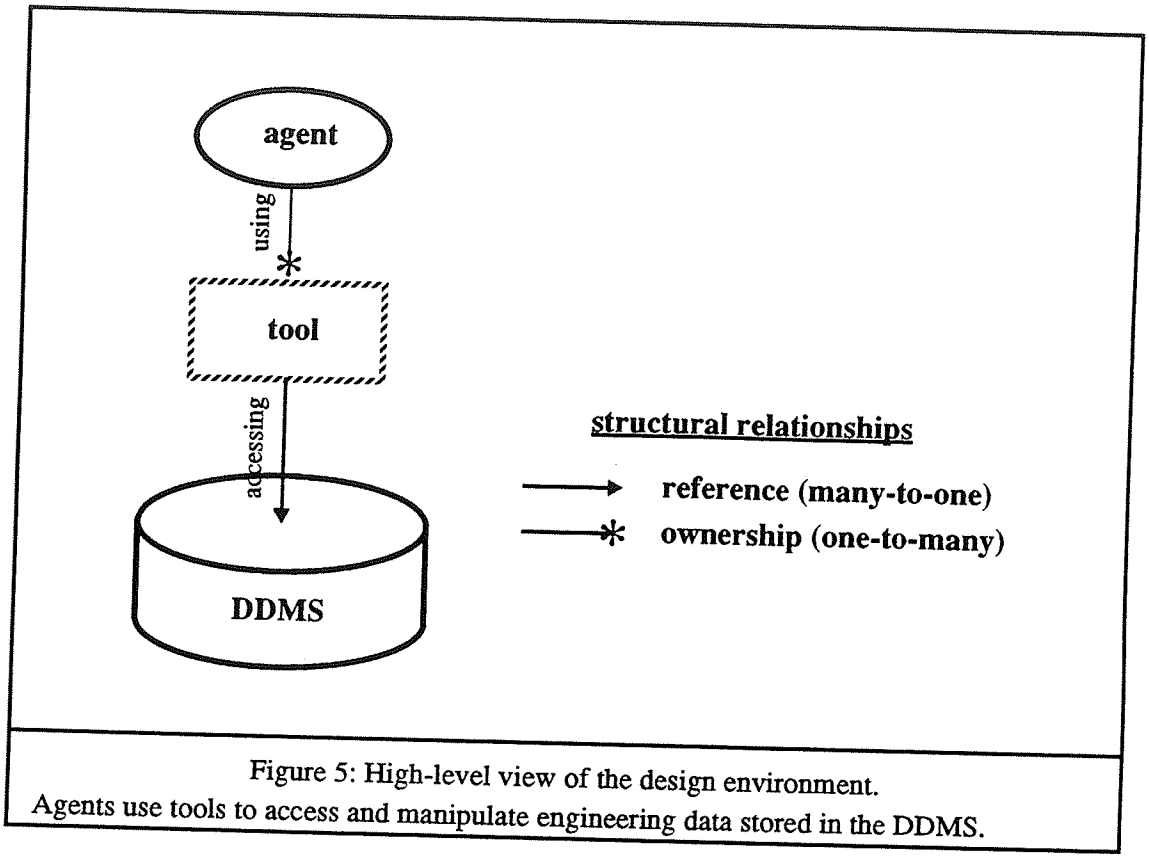

Figure 5: High-level view of the design environment.
Agents use tools to access and manipulate engineering data stored in the DDMS.

Section 1.D.1: Agent

In this thesis, an **agent** is a human who is involved in the design of one or more artifacts. An agent either helps complete or manages the completion of designs, and is either an **engineer** or a **supervisor**, respectively. A permission is a right given by a supervisor to an engineer to update a design object. See Figure 6.

Section 1.D.2: Tool

A **tool** is a program which can be used by an agent to access design data in the DDMS. Tools, unlike the DDMS, provide a user interface to the agents that use them. The DDMS can be used only indirectly through tools. Tools include programs used by either engineers or supervisors, such as design editors, simulators, circuit extractors, compilers, and browsers. One agent may use multiple tools at the same time.

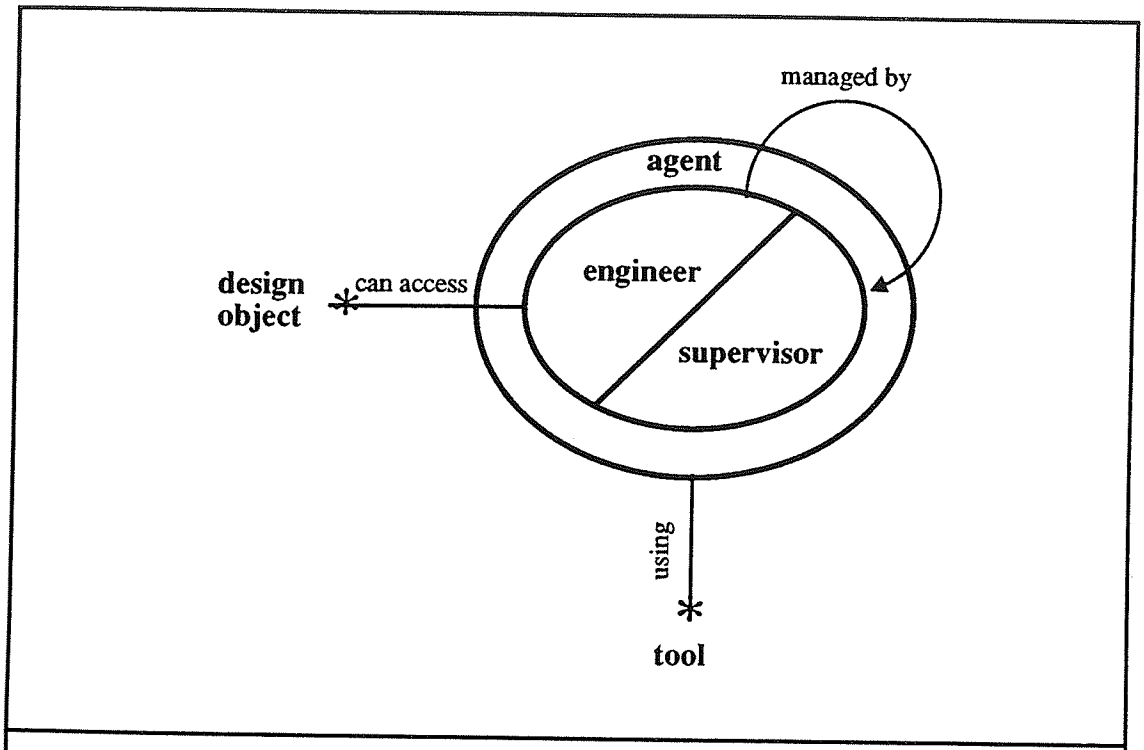

Figure 6: Agents.
Agents are humans that use CAD tools, and are either engineers or supervisors. Each engineer reports to a supervisor. Supervisors assign engineers permissions to update design objects.

Note that there is a difference between a program and a process, which is a running instance of a program. Likewise there is the difference between a tool and a running instance of a tool. In the remainder of this document, however, the term **tool** will be used for either, and context will disambiguate the meaning.

Tools are independent. They jointly access the DDMS, but each tool is unaware of other tools' existence except indirectly through messages received from the DDMS (or more precisely, from the Change Manager within the DDMS, as will be explained in Chapter 3) as a result of the actions of other tools. Unfortunately, there does not exist an algorithm to determine whether an update to the design constitutes progress toward a superior or more complete design. Thus there is no way to enforce that tools are *cooperating* and collectively

operate in a constructive fashion. Because of this, it is the responsibility of the human agents to ensure that the concurrent use of tools on shared design data is a cooperative effort.

A tool that supports concurrent design consists of **application code**, which implements the tool's functionality, plus the Change Manager. As mentioned earlier, the Change Manager provides support for tools to concurrently access shared design data; the support provided is elaborated in Chapter 5. The application code, in order to make use of the Change Manager and thus perform suitably in the concurrent design environment, must obey a set of rules which make it **well-behaved**, as explained in Chapter 6.

The Change Manager within the tool manages an **annotated object cache**, which is the tool's local object workspace. Designs are dynamic entities. As will be explained in Chapter 3, to achieve concurrent design requires that each application stay informed of changes made by other tools to objects which it is in the process of updating. In addition, an application may request to be informed of changes to the design status. The Change Manager in a tool achieves this by placing triggers on objects in the cache [Dayal 90]. Not all updates to objects in the cache or changes to the design status are important to every application, and since the design of the Change Manager is independent of, and thus doesn't understand, the semantics of the individual applications, it cannot by itself determine of what events an application needs to be informed. For this reason each application must specify **application interests** to the Change Manager. An application interest identifies some set of events in which it is interested.

When an event occurs which matches an interest, the application which registered the interest is sent an **internal notification**. If a different tool updates an object that has been cached, the Change Manager in the tool with the invalid cache is sent an external notification which is to be incorporated into the cache. See Figure 7. Application interests and no-

tifications are described in detail in Chapter 5. The interaction of the components within a tool is the subject of Chapter 6.

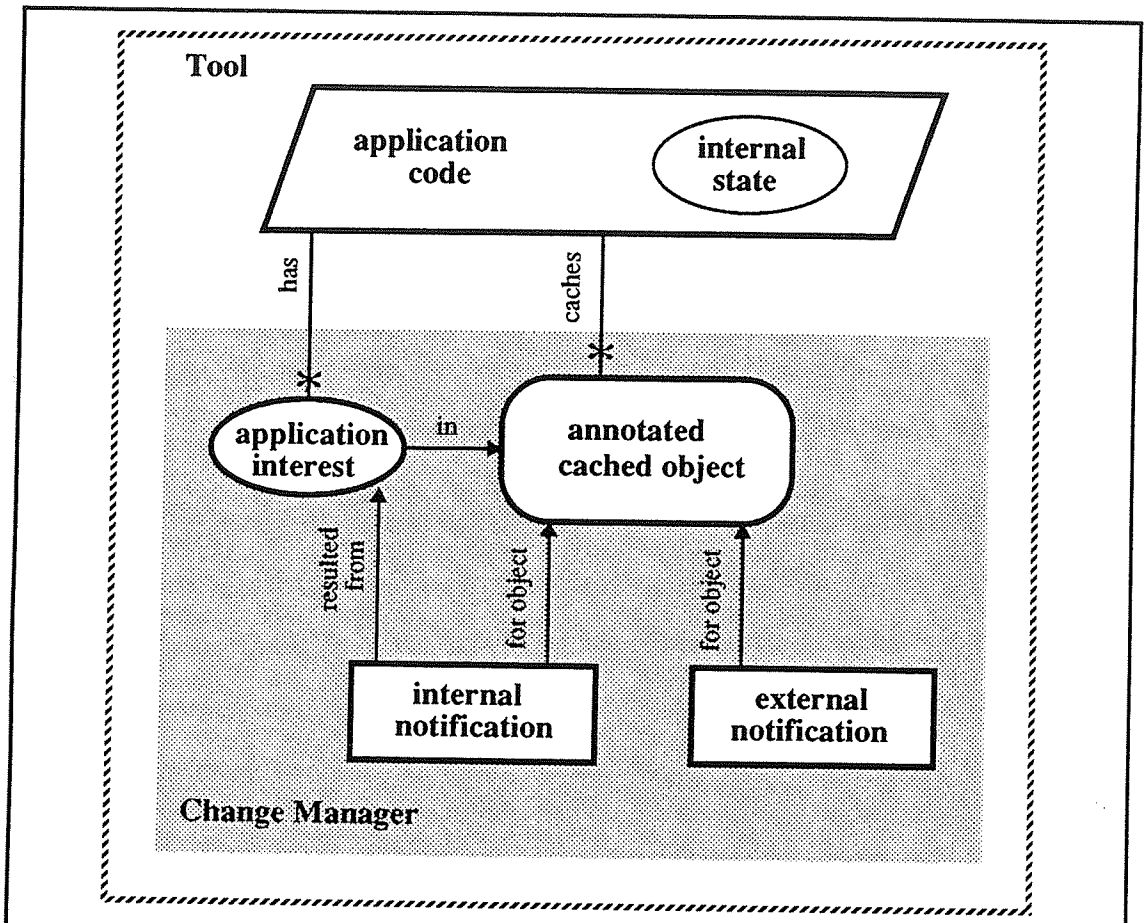

Figure 7: The major components of a tool that supports concurrent design. Application code implements the tool's functionality. The annotated object cache holds uncommitted updates to objects and bookkeeping information. Application interests specify which changes to the object cache the application code wishes to track, and internal notifications are queued when such a change occurs. External notifications are a result of changes to objects in the DDMS by other tools.

Section 1.D.3: Design Data Management System

The database system used in a design environment we call a **design data management system**, or DDMS. The DDMS offers tools the ability to access and make persistent changes

to design data. Tools, rather than the DDMS, offer a user interface for engineers. Thus the DDMS needs to provide only a programmatic interface to design data for use by tools. As mentioned in Chapter 1, use of "*the* database" or "*the* DDMS" is not meant to exclude multi-databases or distributed databases, but rather to refer to the aggregate functionality of the database system being used.

The **DDMS** consists of an object-oriented **data store** with associated **schema** as a substrate, plus additional components which are part of the Change Manager: **workspaces**, **tool interests**, **design status**, and **conflict records**. See Figure 8. Chapter 3 will detail in what ways a traditional database system is inadequate for use as a DDMS, thus identifying the need for the Change Manager. The interaction of the components within the DDMS is the subject of Chapter 6.

Objects

All persistent design data are stored in and accessible from the DDMS. In order both to control access to portions of design data and to make manageable the amount of data which is transferred between tools and the DDMS, design data are divided into a large number of interconnected **objects**. Objects are used to represent designs, subdesigns, and components within the DDMS.

Our use of term **object** coincides with the use of that term in object-oriented circles [Heiler 87][Spooner 85], in that an object has a type, an identity, an internal state, and a programmatic interface to access and change that state. Objects in the DDMS possess other characteristics and are interrelated. The object model used by the Change Manager is presented in detail in Chapter 2.

Workspaces and the Check-out/Check-in Protocol

Workspaces are work areas in which tentative updates are made. When those updates are no longer tentative, they are **committed**. Before a tool can update a design in a workspace,

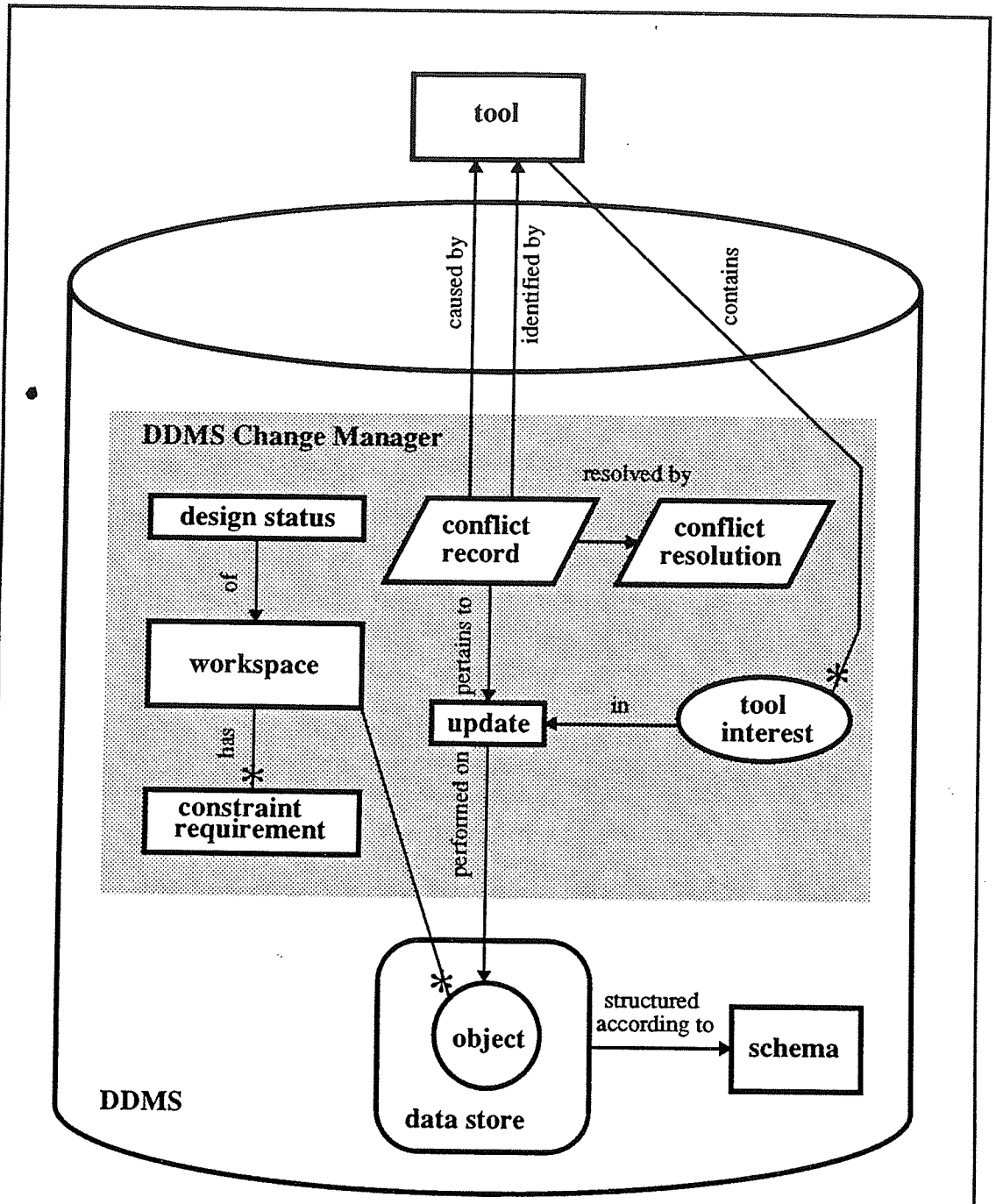

Figure 8: The components of the Design Data Management System.
The DDMS consists of an object-oriented data store and schema plus the DDMS Change Manager, which contains workspaces, tool interests, design status, conflict records, and conflict resolutions.

the tool must **check-out** that design into that workspace. This action indicates an intent of the tool to update that design in the workspace. Intent to update a design is released when the object is **checked-in**.

Although many engineers may have permission to update the same design object, that object can be checked-out in at most one workspace at any time. Thus if two engineers wish to update some design object at the same time they must do so within the same workspace. Workspaces and their semantics are described fully in Chapter 3.

Design Status

As mentioned in Section 1.C.2, knowledge of design status can assist engineers in coordinating their design efforts with those of other engineers. Design status is maintained by the DDMS and made available to all tools.

Tool Interests

Before committing changes to the DDMS, an application in a tool makes changes to cached copies of the objects it is updating. This cache is managed by the Change Manager in the tool. The cache may grow stale, in which case the tool must be informed of the ways in which the cache has grown stale. The Change Manager in the DDMS uses **tool interests** to record what objects have been cached by what tools, employs triggers to make objects in the object store **active**, and sends external notifications to tools when the objects which they have cached are changed by other tools.

Conflict Records And Conflict Resolutions

When an actual conflict is identified, that is, when an agent or tool disagrees with an update made in some workspace, a **conflict record** can be created which references the offending and offended agents, the update which caused the conflict, and the tool which performed the update. Conflict records are appended to the **conflict log**. At some future point in time,

a **conflict resolution** indicates what action was taken on behalf of the conflict. Conflicts are described further in Section 3.B.9.

Part 1.E: Related Work

This part of the chapter surveys research related to that presented in this thesis.

Section 1.E.1: The Use of Databases in CAD

Commercial and academic efforts are underway to investigate how to apply database technology to CAD environments. Below are two examples of the type of research being performed in this area:

U.C. Berkeley

The Oct project at U.C. Berkeley is an early attempt which provides limited database support for CAD tools [Harrison 86]. Oct supports a very simple data model: designs consist of a set of **cells**; each cell is a hierarchy of **facets**. A cell represents a view of a design, and facets represent components of the design in that view. The interpretation of data in facets is left to tools, and is known as Oct **policy**. The Oct project includes some tools which display and manipulate design data according to the Oct policy. Oct does not offer workspaces which encapsulate updates made by CAD tools. Oct does offer one programmatic interface through which all design data may be accessed.

Delft University of Technology

The NELSIS integrated circuit design system developed at Delft University of Technology uses a custom object-oriented design database as a foundation for its design environment [van der Wolf 88]. The database offers a detailed integrated data model, with ownership and reference relationships and consistency constraints in the schema, and a programmatic interface for accessing design data. The database gives tools two levels of workspaces—

public and private—and makes knowledge of the design status available to tools. The database guarantees referential activity but does not support active data or concurrent updates to design data by multiple tools.

Section 1.E.2: Frameworks for CAD

There exist several ongoing projects to build frameworks into which any number of tools may be integrated and work together. The projects differ somewhat in the amount of alteration required of existing tools for them to be integrated into the framework. This section describes two important projects of this type.

CAD Frameworks Initiative

The CAD Frameworks Initiative is a coalition of several vendors of CAD tools, vendors of object-oriented databases, and some universities. CFI is a commercial effort to define standard file formats for electronic design data. The current emphasis of CFI is to achieve the immediate but limited integration of tools which is possible without changing existing tools in use today. A longer-term goal of CFI is to standardize data models and database interfaces which will be used by tools that are developed in the future.

U.S. Air Force

Faced with rising costs of increasingly complex avionics, the U.S. Air Force initiated in 1985 the Engineering Information Systems project [Wiederhold 85][Wiederhold 89]. The EIS project is an extremely ambitious project whose goal is to integrate all information related to the design, production, and maintenance of electronic systems. In other words, the entire life-cycle of electronic artifacts is being considered. In addition, an attempt is being made in the EIS project to provide evolutionary migration paths from existing techniques, tools, and data formats into the integrated environment.

The EIS project unfortunately does not use database technology which has appeared only recently, such as **active data** (described in Section 1.E.5). For this reason, we believe that EIS project will be obsolete if it is completed. Unlike the evolutionary approach of the EIS project, this thesis presents a revolutionary approach, which, this thesis argues, is needed in order to achieve design-centricity in the design environment.

Section 1.E.3: Groupware Systems

An area of active research is Computer-Supported Cooperative Work, also known as **groupware**. Groupware provides automated support for communication and collaboration of a team of people who interact in order to achieve some common goal. Most software systems only support the interactions between a user and the system. The goal of groupware systems is to offer users a way to stay aware of progress in the context of the entire team without being obtrusive.

Microelectronics and Computer Consortium

The Grove project at MCC is pursuing research in groupware systems, and has constructed a prototype shared editor [Ellis 89][Ellis 91]. Users of the editor may be physically distributed, and the views of data which they see may be somewhat stale. By default, all users are allowed to see and edit all documents, even at the same time. Experience with the shared editor has provided five important insights into collaborative computer-based activities:

1. Concurrent work occurs naturally and spontaneously when the restriction that only one person can access a document at any given time has been removed.
2. Concurrent work can be confusing at times, but conflicts are surprisingly infrequent.
3. Learning the strategies of, and acquiring knowledge from, other team members is a natural consequence of concurrent, collaborative activities.
4. Members of a team become familiar with more aspects of the result when they work collaboratively, than if they had worked independently on well-partitioned tasks.

5. The fact that many people participate to achieve a shared goal tends to improve the overall quality of the result.

Section 1.E.4: Transaction Models

Other research has concentrated on providing support to maintain aspects of consistency of designs in the CAD environment. This section describes four such projects:

Hewlett-Packard Labs

Database research at H.P. Labs has investigated the use of **notify locks** and optimistic concurrency control to maintain cache consistency [Wilkinson 90]. Notify locks cause notifications to be sent to interested processes when particular data are updated. The processes use the information contained in notifications to make their caches current. Optimistic concurrency control causes transactions to be aborted if data in the read set of a process are disturbed before the process is able to commit its updates.

This technique is similar to that used by the Change Manager which is described in this thesis, whereby object caches with CAD tools are kept current. An important difference, however, is the support which is provided by the Change Manager to enable a process (i.e., CAD tool, in this case) to continue operation despite updates made by other processes to its read set.

Brown University

Research being performed at Brown University is investigating the utility of what are called **cooperative transaction hierarchies** [Nodine 90] in the design environment. In this research, each design is partitioned into a hierarchy of components, and each component has a corresponding engineer who is responsible for the correctness of that component. A transaction hierarchy is then created, and represents design activity occurring concurrently on the components. This transaction hierarchy is made "cooperative" by defining rules which

specify what engineers must approve what updates to components before the design as a whole is considered correct.

Two assumptions made in the above research are radically different from those presented in this thesis. These assumptions limit the utility of cooperative transaction hierarchies:

1. Cooperative transaction hierarchies are isomorphic to the hierarchies of components in designs. The workspace hierarchy offered by the Change Manager, however, need not be similar to the hierarchy of components in any design. This permits the workspace hierarchy to represent organization aspects of a team of design engineers, wherein any number of people can share responsibility for the correctness of any one component.
2. Cooperating transaction hierarchies require that a design's hierarchy of components be chosen when work on the design commences, and that the partition be fixed throughout the design process. The Change Manager, on the other hand, permits dynamic repartitioning to occur.

University of Texas

Researchers at the Princeton University and the University of Texas are studying sets of database updates which they call **compensating transactions** [Garcia-Molina 87][Korth 90]. A compensating transaction is a set of updates which makes adjustments to the state of the database so that a process can continue operation even though the data in its read set have been modified by other processes. An algorithm to automatically generate compensating transactions is developed in the special case of commutative updates to the database. This research is particularly relevant to the medium level of coordination between CAD applications and the Change Manager, as presented in Section 6.C.2.

University of California at Berkeley

Researchers at U.C. Berkeley have developed a mechanism called **change management** in which a new version of a component is created each time it is checked-out for update. A

system of version propagation is proposed in which the check-in of new versions of a set of referenced objects results in the creation of new versions of referencing objects; this process applied recursively up the design hierarchies [Katz 87]. (Note that their meaning of the term "change management" differs sharply from that of the term "Change Manager" used in this thesis.)

Interpretation of a rule base determines how far up within the design hierarchy new versions of parent designs are to be created. These new versions will incorporate the new versions of subdesigns. The research proposes that this recursive creation of new versions terminate in certain situations, for example (1) when an object has been declared "independent" of changes to subdesigns, or (2) when the interface of the new version is incompatible with what is required of a referencing parent design.

When a version is referenced multiple times, there may be ambiguity as to which references to update to point to the new version; in this case the engineer disambiguates the references prior to check-in. This research gives no consideration to shared updates by multiple engineers to the same or related design objects.

Section 1.E.5: Active Data

A recent area of database research involves the use of **active data**. Active data can cause predefined side effects to occur when they are updated. Active data are relevant to this thesis because the Change Manager uses active data to keep tools mutually aware of each other's updates: the Change Manager sets the side effect associated with active data to be the delivery of asynchronous update notifications to tools.

Hewlett-Packard Labs

An extension to the Iris object-oriented database being researched at H.P. Labs is the **database change monitor** [Risch 89][Risch 91]. A database change monitor offers translation of low-level updates to a high level of abstraction, and gives database clients the ability to track changes to the results of queries, not merely simple state changes. This capability could be used by the Change Manager in its interest matcher, which is described in Section 5.B.6.

Chapter 2

The Object Data Model

Part 2.A: Introduction

The DDMS manages structured design data and makes them available to multiple tools. A database can implement any of several types of data models. The DDMS implements an **object-oriented type of data model**, in which design data are broken into a collection of interrelated **objects**. Objects represent entities in the real world or in the abstract design world [Du 87][Wiederhold 86a]. It is important to distinguish between the type of data model, such as relational, hierarchical, or object-oriented, and a specific data model which, in the case of an object-oriented, determines the types and interrelationships of objects in the database [Korth 86][Wiederhold 86c].

This chapter describes the characteristics of objects, 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 elaborates the semantics of derived and computed slots, enumerates what operations can be performed on objects, offers two detailed examples of simple schemata for design data, and concludes with a discussion of ways in which the object model presented might be extended.

Part 2.B: Characteristics of Objects

Objects have three important characteristics: identity, type, and state. Each of these is discussed in the sections below.

Section 2.B.1: Identity

Each object has an identity and corresponding **object identifier** or **OID**, which is guaranteed to be different from the OIDs of all other objects. The OID is used as a handle with which a client of the DDMS (such as a tool) can reference and access the corresponding object.

Objects are created and destroyed dynamically. The lifetime of an object is independent of the lifetimes of other objects, except in the case of the object being owned by another object; this case will be explained later in this chapter.

Section 2.B.2: Object Type

An object, when it is created, is specified to be of a particular **object type**. The type of an object is fixed for the lifetime of the object and thus cannot be changed. Objects of a given type are called **instances** of that type. An object type determines the structure of instances of that type, that is, what slots the objects of that type have, and represents the category to which the entities represented by those instances belong. Layout, Module, Rectangle, and Schematic are examples of object types. A field names a value or **instance variable** which is specific to an object.

Section 2.B.3: State

An object type describes the structural aspects common to all instances of that type. Objects of the same type are specified and differentiated by their **state**. The state of an object is

maintained in a number of named **slots**. Slots are explained in detail in the next part of this chapter.

Part 2.C: The Object Schema

The particular data model offered by the DDMS will depend upon the engineering domain chosen and upon design decisions made by the person who defines the data model. The DDMS can support any one of a number of data models. The data model in use, that is, the object types and the structure and types of their slots, is described by the **schema**. The syntax of a schema is defined by ten production rules below in Backus-Naur Form (BNF) [Backus 60].

1. *schema* := *objectTypeDecl* ...

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

2. *objectTypeDecl* := *objectTypeName* [*slotDecl* ...]

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

3. *objectTypeName* := identifier

   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.

4. *slotDecl* := *slotName* : *slotType*

   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.

   Although a slot of one object can be referenced by other objects (described below), the slot is owned by exactly one object and is not shared, and updates to a slot must be done through the object which owns it.

5. *slotName* := identifier

   Slots are named by identifiers. Slot names are unique within an object type.

6. *slotType* :=

   typeName |

   computed *typeName computedSlotSpec* |

   derived *derivedSlotSpec*

   A slot's value either can be assigned and of a specified type, can be the result of a potentially complex computation applied to the values of other slots (a **computed** slot), or can be equal to another object or a slot of another object (a **derived** slot).

7. *typeName* :=

   primitiveType |

   objectTypeName |

   set *typeName* |

   ref *objectTypeName*

   A slot's value can be specified to be either a primitive type (described below), an object type, in which case the value is a subobject owned by the object which owns the slot, a set of values of a specified type, or a reference to another object of a specified type.

   There are important differences between a slot's value being a subobject (or a set of subobjects) and its value being a reference to another object: In the former case, the lifetime of the subobject is tied to that of the owning object in that the subobject is created or destroyed when the owning 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. Subobjects and object references are examples of the "ownership connection" and "reference connection" [Wiederhold 83], respectively.

Another difference between subobjects and referenced objects involves different interpretations of timestamps which indicate when a object's slots were last changed. Timestamps are discussed fully in Chapter 6.

8. *primitiveType* :=

   Boolean |

   integer |

   string

There are three primitive types in DDMS: **Boolean** (true or false), **integer** (numerical), or **string** (array of characters). It should be noted that these types are merely examples. Other types might be added and are absent only for the sake of simplicity.

9. *computedSlotSpec* := { *slotName* ... }

In some cases, the value of the slot, which represents some aspect of the object, depends upon the values of other slots in a way that requires some arbitrary computation. The DDMS does not attempt to keep these computed values current. Such computations are the responsibilities of tools. The schema of the DDMS, however, has *computedSlotSpec* entries which indicate upon which slots the computed slots depend. Derived slots and computed slots are explained later in this chapter.

10. *derivedSlotSpec* :=

    slotName . *slotName* |

    slotName *

As stated earlier, the value of a slot can be copied from subobjects or referenced objects. The *derivedSlotSpec* specifies how that value is to be obtained. A derived slot may have the value of the slot of a subobject, which, in the case of a set-valued slot, would result in a set of values (and is similar to the projection operator Π in relational algebra), or may be the result of following a reference to another object.

Part 2.D: Relationships Among Objects

Although design objects are independent entities with their own separate identities, two objects can be related to one another in any of four ways: ownership, reference, version, and alternative. This part of the chapter will discuss those four types of relationships.

Section 2.D.1: Ownership

It is possible for an object B to be nested within another object A, as defined by the data model. In this case the nested object B is said to be a **subobject** of A that is **owned** by A. An object can be a subobject 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 members cannot be determined in advance; an example is a design which owns some number of components.

An object which owns other objects can itself be nested in an object. Thus, recursive nesting of objects can give objects an hierarchical structure. Ownership is acyclic. A given object can have at most one owner, and its owner (if it has one) is fixed for the lifetime of that object. As mentioned earlier, the lifetime of an owned object is tied to its owner: when an object is created, subobjects are also created (except in the case of a set-valued slot, whose initial value is the empty set Ø); when an object is destroyed, so are its subobjects (and in the case of a set-valued slot, all subobjects in that set). These rules apply recursively.

Because a subobject is so closely tied to its owning object, it may be considered part of that object. An object which is not owned by another has a lifetime independent of other objects. It may thus be thought of as "top-level" or "free-standing" and is given a special name: **design object**.

Section 2.D.2: Object References

When an object is owned by another, it is accessible through a slot of the owning object, and its lifetime is tied to that owning object. There is another way to make an object accessible to another object: through a **reference**. A reference is a handle to an object which can be stored in the slots of another object. If a slot of design object A or a slot of a subobject of A references design object B or a subobject of B, then A is said to reference B, denoted $A \rightarrow B$.

References are useful because they permit sharing of information. In a design 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. See Figure 9.

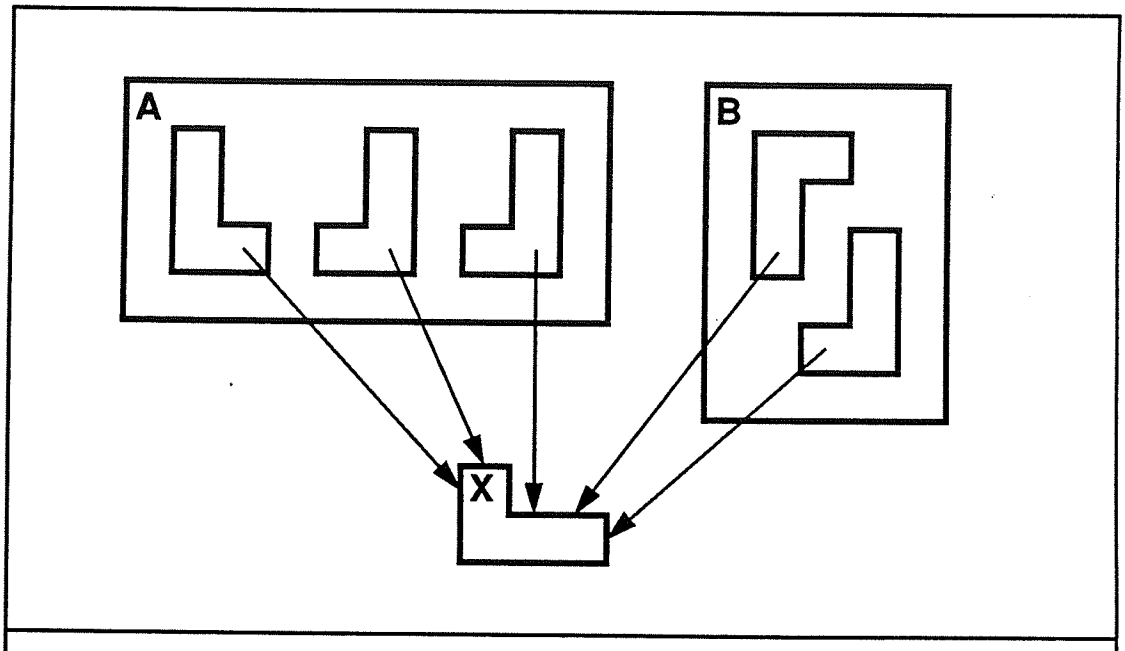

Figure 9: Object references.
Designs A and B use references to incorporate multiple instances of design X into their designs. The instances, because they are subobjects of the parent design, are destroyed when the parent is destroyed.

When an object with references to other objects is destroyed, those references are destroyed. "Referential integrity" is enforced, however: an object cannot be destroyed if there are references to it. Unlike ownership, an object can have any number of references to it. Object reference may be cyclic, as in Figure 10.

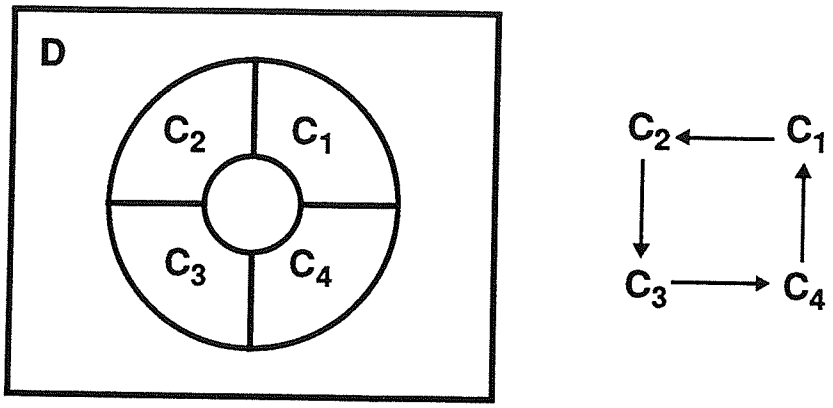

Figure 10: Cyclic references.
In this figure, design D is a ring. This can be represented in the object model by having each of the components of D reference its counterclockwise neighbor.

Dependency

Let X, Y, and Z be design objects. Then we define the "depends upon" relation $\Rightarrow$ between design objects as the transitive closure of "references" $\rightarrow$:

$X \Rightarrow X$

$X \Rightarrow Y$ and $Y \rightarrow Z$ implies $X \Rightarrow Z$

Note that for a set S of objects that reference each other circularly, $X \Rightarrow Y$ for each $X, Y \in S$.

Given the definition of $\Rightarrow$, we now define the "sources of X" Src(X) and the "dependents of X" Dep(X) as:

$$Src(X) \equiv \{ \text{ all Y such that } X \Rightarrow Y \}$$

$$Dep(Y) \equiv \{ \text{ all X such that } X \Rightarrow Y \}$$

These definitions will be useful later in this chapter when derived and computed values are discussed, and in other chapters.

Section 2.D.3: Versions and Design Elements

Besides not being owned by another object, design objects are special in another way: every design object belongs to a set of design objects which are related in that they are **versions** of the same **design element**. Versions of a design element form a linear sequence in which each version except the first has a **predecessor** and each except the last has a **successor**. A version is considered to be "derived from" its predecessor (but this should not be confused with the notion of a computed value being derived from its source slots).

Only the latest version of a design element can be updated. The sequence of versions $X_1 \rightarrow X_2 \rightarrow ... \rightarrow X_n$ for a design element X corresponds to a progression of attempts to implement a particular piece of functionality; newer versions are *corrections* or *enhancements* of earlier versions. The last (and most recently created) version of design element X is the **latest version** of X; other versions are called **obsolete**.

Version versus Alternative

The concept of **version** should not be confused with that of **alternative**. An alternative is a different design element, which shares similarities of functionality to some degree with a given design element, but which has a different intended application. For example, an edge-triggered flip/flop may be considered an alternative to a latch, and may have been designed by altering some version of a latch. When a designer uses an instance of a design as a com-

ponent in a design, the designer must choose whichever alternative is appropriate. The concept of alternative, although useful in a design database, is not important to the research presented in this thesis and will not be considered further.

Part 2.E: Computed and 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 S is called a **derived slot**. The object model presented in this thesis divides derived slots into two categories: **computed slots** and **derived slots**. This part of the chapter explains the semantics of computed and derived slots. Part 2.G gives two examples of the use of computed and derived slots.

Computed Slots

The value of a slot may depend upon the values of other slots in a way that involves an arbitrary amount of computation. Such a slot is called a **computed slot**. An example of a derived value is the bounding rectangle of a circuit design, which is a relatively simple computation based upon the dimensions of its components and the size and placement of the interconnect.

The schema within the DDMS designates a *computedSlotSpec* for each computed slot, which is a list of slots upon which the value of the computed slot may depend; see production rule #9 in Part 2.C. These slots are called **source slots**. If one or more source slots changes, the computed slot is said to become **void**, that is, it requires a recomputation in order to acquire a current value. See Figure 11.

Note that *the schema does not specify a function whose execution will recompute the value of the computed slot*. This is an important point. The schema and DDMS do not require that a particular function or tool be used for computation. That is because computation is out-

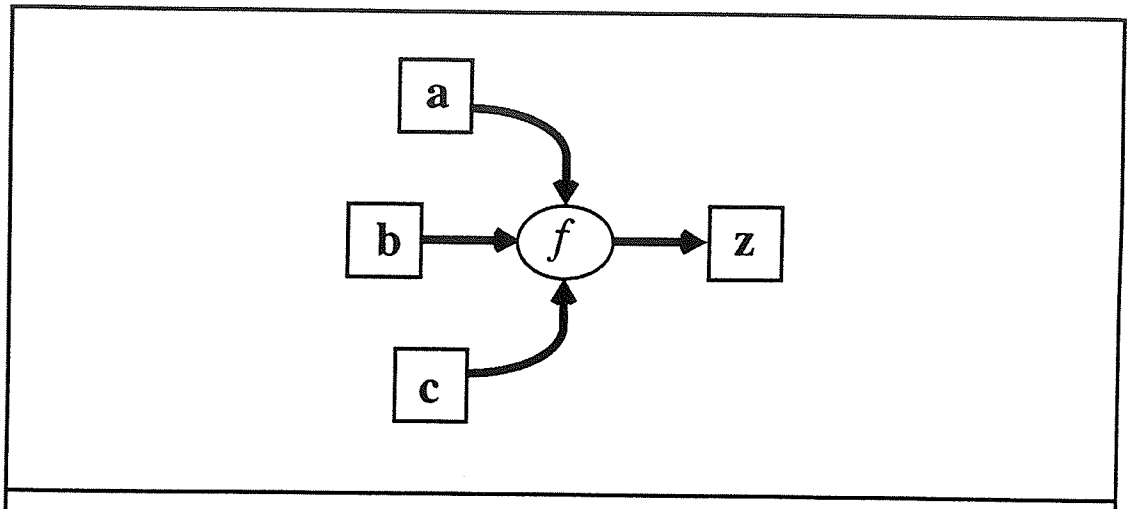

Figure 11: Computed slots.
The value of slot **z** is a function of the values of slots **a**, **b**, and **c**. The schema specifies the dependency of the computed slot on other slots, but the function to recompute the computed slot is not specified; that computation is the responsibility of one or more tools.

side the scope of the DDMS and in the realm of CAD tools. Tools are responsible for computation. A supervisor may require that the engineers use a particular tool to perform design-rule checking, for example [Bhateja 87], but that is design *policy* and is outside the scope of this thesis.

In some proposals for next-generation design environments, which tools to use for validating consistency of the design is part of the schema [Hall 88]. This approach is useful for automatically invoking validation tools but is less accommodating to engineers' preferences to using particular tools.

Part of the integrated data model offered by the DDMS are **constraints** which represent aspects of consistency of a design. Constraints may be met or not; their state depends upon the outcome of **validation functions**. When met, constraints ensure some level of "correctness" of the design. When data are changed, constraints which depend upon those data may no longer be valid. Which constraints are present in the data model is specific to the engi-

neering domain; design rule checking, stress analysis, and type checking are examples that would be found in electrical, mechanical, and software engineering, respectively.

Components of a design, since they collectively implement the design, are interrelated. Since different engineers may work on different but related designs, the global consistency of a parent design, which uses instances of those designs as components, may be disturbed. Validation functions in tools, as described above, are applied across components when the consistency of the parent design is to be ascertained.

Design constraints and consistency checks are modeled as computed slots; the validation functions of the constraints are the functions which compute the values of these slots. In the simplest case such a slot has a Boolean value which indicates whether the constraint is satisfied. More generally, a constraint may contain a complex value, such as a list of design rule violations.

Derived Slots

The schema can specify that the value of a slot is to be the same as that of another slot, or the subobject(s) of a slot, or the result of following a reference to another object, or some combination of the preceding. The *derivedSlotSpec* specifies how that value is to be obtained.

Since copying a value, extracting a subobject, and dereferencing are inexpensive operations in an object-oriented data store, a tool can quickly recompute the value of a derived slot when one of its source slots changes. Thus derived slots, unlike computed slots, stay current with respect to their source slots. Note that the decision that particular computations are "inexpensive enough" to be automatically recomputed is not clearcut. An extension to the object model presented here might include additional operations, as described in Section 7.C.2.

Part 2.F: Operations on Objects

Design 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 3. The operations permissible on a slot depend upon the type of the slot. This part of the chapter presents those operations that create and destroy design objects, describes what updates on slots of objects are permissible for each type of slot, then explains how updates to one object may affect the state of another.

Section 2.F.1: Operations on Design Objects

Creation

A design object of a specified type can be created. The new object can be created as the latest version in a chain of versions of a design element, or as the first version of a new design element. The newly-created object is given a unique object ID; its slots assume default values.

Destruction

A design object can be destroyed. When a design object is destroyed, all its subobjects 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 design 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 destroyed.

Undestruction

A design object can be undestroyed, which means that the effect of previously destroying the object has been undone, with the restriction that references to nonexisting objects are

nullified. Undestroying is similar to creating a new object, except that the identity and state of the undestroyed object are the same as those before the object was destroyed.

Section 2.F.2: Operations on Slots

Primitive Slot

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

X.S ← *v*, where *v* is of the appropriate primitive type.

Subobject

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

Set of Subobjects

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

create new member in X.S

destroy member Z ∈ X.S

undestroy member Z ∈ X.S

update Z ∈ X.S, as described in this section.

Reference

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

X.S ← ⊥ (undefined), which nullifies any existing reference, or

X.S ← ^Z, which assigns a reference to object Z to slot S.

Computed Slot

Two types of operations may be performed on a computed slot S of object X:

X.S $\leftarrow \downarrow$, which marks the value of the computed slot as void, or

X.S $\leftarrow \uparrow$, which marks the value of the computed slot as valid, or

X.S $\leftarrow v$, where v is of the appropriate primitive type.

The latter two operations would be performed only by an application in order to recompute the computed slot based upon current values of the source slots.

Derived Slot

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

Section 2.F.3: Object State

The state of an object consists of the values of its slots. The value of a derived slot may involve following a reference to another object, and the value of a computed slot may depend upon the value of a derived slot. Thus, updating a slot in one object X may affect the values of derived and computed slots in other objects; this can have a cascading effect. But the state of object Y can be affected by updates to object X only if $Y \Rightarrow X$ or, equivalently, only if $Y \in \text{Dep}(X)$.

Part 2.G: Example Schemata and Design Objects

This part of the chapter gives two examples of schemata and associated objects. The first example is from the domain of electronic CAD, the second from the domain of software CAD (also called computer-aided software engineering or CASE). Other schemata would be used for other domains [Eastman 91].

Section 2.G.1: Electronic CAD: Integrated Circuit Layout

This section provides an example schema and objects for an integrated circuit layout. Each layout consists of a set of rectangles plus a set of components. Each component is an instance of a layout. This example makes use of three object types, three computed slots, and five derived slots.

Schema

```
Layout
   [
      contents: set Rectangle
      localBBox: computed Rectangle
         { contents }
      components: set LayoutInst
      subDesignRefs: derived components.layout
      subDesigns: derived subDesignRefs *
      componentsX: derived subDesigns.x
      componentsY: derived subDesigns.y
      componentsBBox: derived subDesigns.compositeBBox
      compositeBBox: computed Rectangle
         { localBBox, componentsX, componentsY, componentsBBox }
      designRulesMet: computed Boolean { contents, subDesigns.designRulesMet }
   ]
```

A layout contains a set of rectangles and uses instances of other layouts as components. Each layout has a local bounding box that is computed from the rectangles, and a composite bounding box that is computed from the local bounding box and from the locations and bounding boxes of the components. Any change to a rectangle marks the local bounding box void; any change to the local bounding box or the bounding box of a component marks the composite bounding box void.

Each layout also has a Boolean-valued computed slot that indicates whether design rules are met; this is an example of a constraint. In this example any change to a rectangle or component would mark this constraint void.

LayoutInst
[
x, y: **integer**
layout: **ref** Layout
]

A layout instance is a reference to a layout plus a designation of that layout's position.

Rectangle
[
x, y, w, h: **integer**
material: poly/diffusion/metal
]

A rectangle has width, height, and location and is of a specified material.

Objects

Figure 12 shows layout A which uses instances of layouts B and C as components. Layout A contains ten rectangles, seven of which it obtains from its components. Layout B contains three rectangles and layout C contains two rectangles.

Section 2.G.2: CASE: Software Modules

This section provides an example schema and objects for software modules in a CASE environment. In this schema, a program is built from subroutines and libraries. 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, two computed slots, and four derived slots.

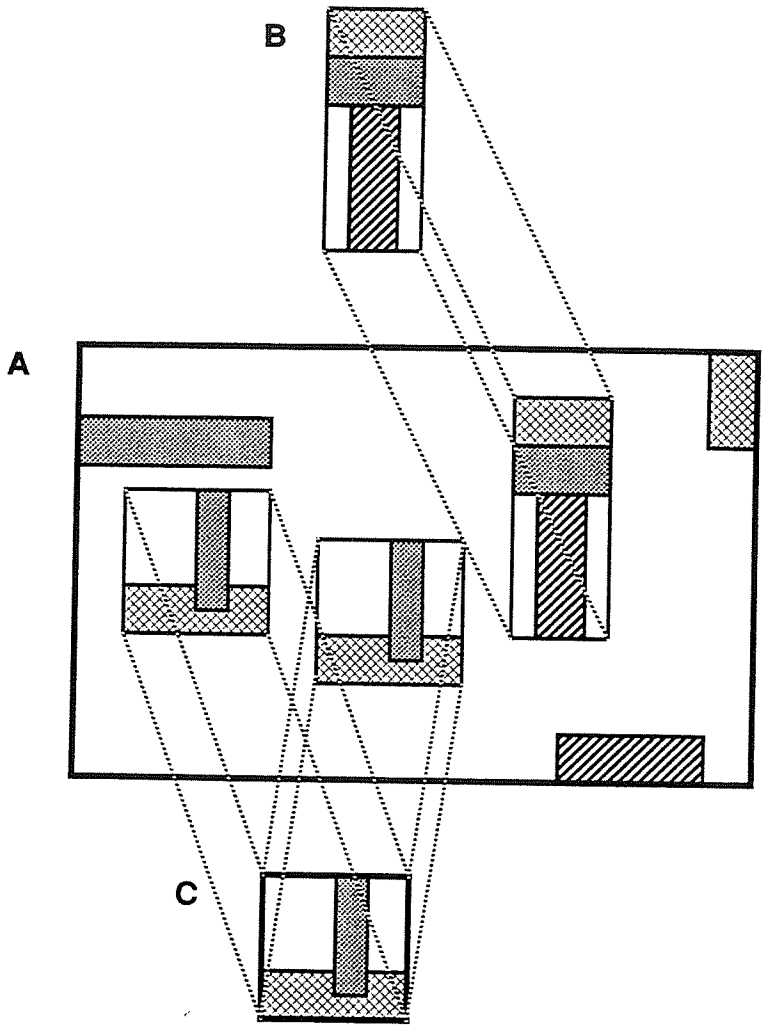

Figure 12: Example objects using the layout schema.
Layout A uses one instance of layout B and two instances of layout C. The bounding box of each layout is a function of the size and placement of its rectangles and components.

Schema

Program
[
subroutineRefs: **set ref** Subroutine
subroutines: **derived** subroutineRefs *
subObjCode: **derived** subroutines.objCode
libraryRefs: **set ref** Library
libraries: **derived** libraryRefs *
libObjCode: **derived** libraries.objCode
executable: **computed bits** { subObjCode, libObjCode }
]

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 marked void and must be recomputed by a linker.

Subroutine
[
srcCode: **bits**
objCode: **computed bits** { 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 void and must be recomputed by a compiler. The "bits" designation for source and object code merely indicates a primitive type of data that have no semantic meaning to the database, and would probably contain ASCII text and machine instructions, respectively.

Library
[
objCode: **bits**
]

A library consists of pre-compiled object code.

This example is simplistic. A better schema for CASE would include structure for subroutines, possibly a parse tree, rather than having two "bits" slots to represent the source and object code for the subroutine. Adding structure would permit updates at a fine granularity

and allow smart compilers and linkers to incrementally recompute object code and executables [Breitbard 68].

Objects

Figure 13 shows programs $Prog_1$ and $Prog_2$ which use subroutines $Main_1$ and $Main_2$ and library LibC; this example uses the programming language C.

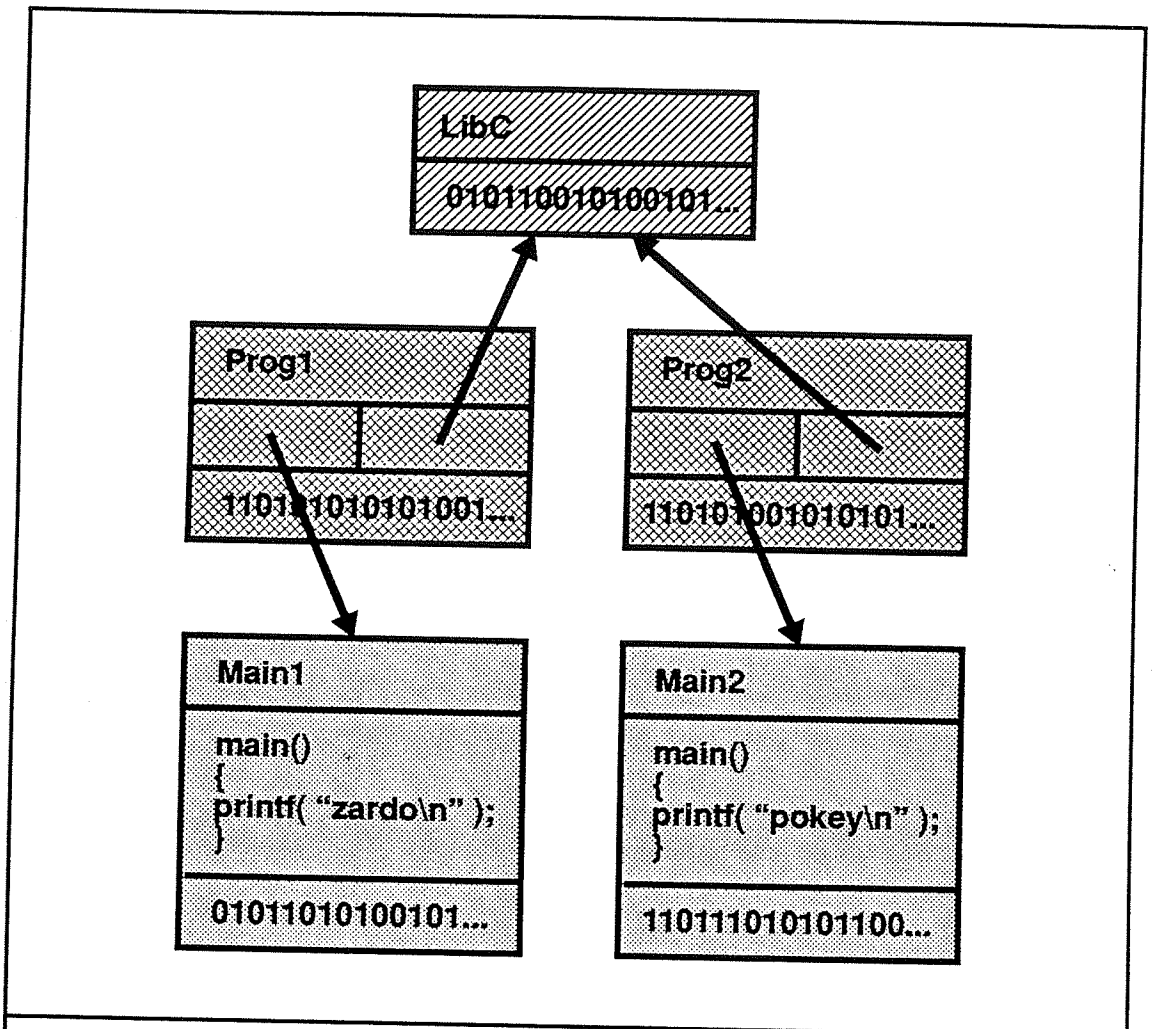

Figure 13: Example objects using the software module schema.
The executable code for program $Prog_1$ ($Prog_2$) is built from the object code from $Main_1$ ($Main_2$) and object code from LibC. If the source code for $Main_1$ ($Main_2$) changes, the object code for $Main_1$ ($Main_2$) and the executable for $Prog_1$ ($Prog_2$) are marked void.

Part 2.H: Extensions to Object Model

The object model presented in this chapter is by no means the only model which is reasonable. It was chosen to be illustrative of characteristics which are useful in the design domain and to serve as a formal foundation for later chapters in this thesis. In particular, any object model chosen should minimally include nested objects, object references, set-valued slots, and computed slots. Extensions to this model may offer other characteristics or capabilities that make the application programmer's job easier or improve efficiency or both. Section 7.C.2 discusses ways in which the object model can be extended.

Chapter 3

Motivation

This chapter motivates research on the Change Manager by describing the characteristics of traditional databases and CAD tools and by identifying the assumptions made when those databases and tools were developed; these characteristics and assumptions limit the amount of concurrency which can exist in the traditional design environment. This chapter also discusses features which databases and tools need in order to support concurrent design, but which traditional databases and tools lack.

The first part of this chapter discusses the traditional design environment and describes how the traditional design environment fails to adequately support concurrent design. The second part discusses features of the next-generation design environment which compensate for weaknesses of the traditional approach. The third and final part proposes the Change Manager as a means to provide the features discussed in the second part, gives a high-level view of the Change Manager, and lists its requirements. Later chapters discuss in detail what services the Change Manager provides for the database and tools, how the Change Manager operates, and what invariants it enforces and degree of consistency it guarantees.

Part 3.A: Traditional Design Environment

This part of the chapter describes the design environment offered by a traditional database and traditional tools.

Section 3.A.1: Traditional Tool Architecture and Operation

A traditional tool consists of application code, internal state, and a translator from and to the data model offered by the database. See Figure 14.

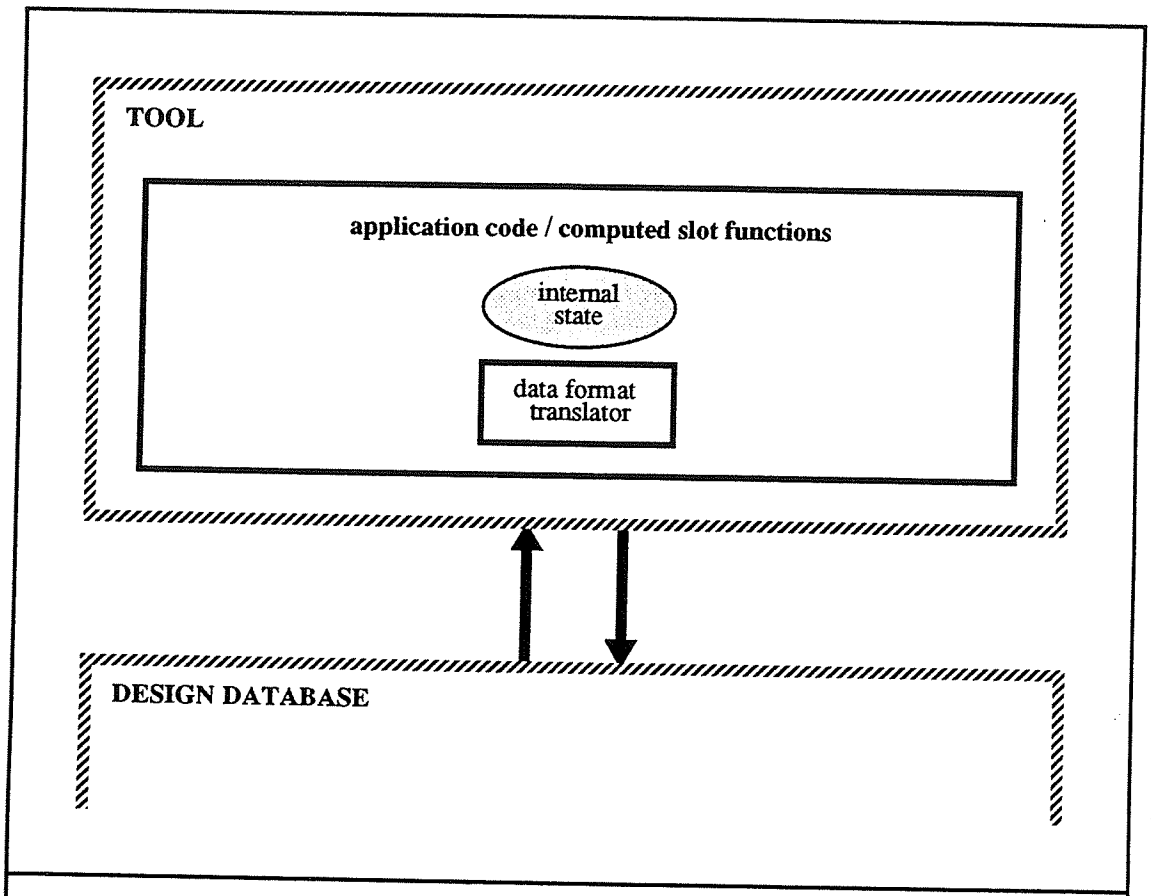

Figure 14: Traditional tool architecture.
The tool loads data from the database and translates it to a tool-specific format. When changes are saved, the reverse translation occurs.

The operation of a traditional tool consists of repetitions of five phases; these phases are described in this section.

Acquisition of Locks

Before a tool can manipulate data, it must secure write locks on the design objects to be updated, and read locks on design objects it will use during the course of its operation. Locks are acquired by use of the check-out protocol described in the next section.

Data Loading

A tool developer uses particular data structures and algorithms in order to achieve efficient operation of that tool. Because different tools perform different tasks, the data structures and algorithms chosen to maximize that efficiency depend upon the tool. The design database offers one integrated data model to tools; design data must be accessed and stored using this data model. No single data model can offer the representation of design data which is most efficient for all tools.

The design database maintains data in some normalized format; tools then interact with the database using the common data model, as defined by the database schema. In the case of a database that offers an object model, this means that the contents of a tool's data structures are derived from the objects read from the database and that updates from a tool to the database must be presented as updates to objects.

After the appropriate locks have been acquired, a tool loads from the design database those design objects which it needs to access or update. Each tool has the responsibility of translating between the view offered by the database and the special-purpose data structures which comprise its internal state. The translation is performed by the tool's data format translator.

Data Manipulation

After data have been translated to a form appropriate for the tool, the design data are manipulated. A tool such as an editor requires a high degree of interaction with the engineer

who invoked the tool during this phase. Other tools such as consistency checkers or simulators can perform their tasks with little direction from the engineer. It may be impossible to predict the duration of this phase of a tool's operation; data manipulation may continue for days or weeks.

Data Unloading

After a tool has manipulated design data to the satisfaction of the engineer who invoked the tool, the internal state of the tool is translated to changes to design objects in the database, and these changes are sent to the database. These design objects then assume their new state in the database.

Release of Locks

After a tool has finished accessing or manipulating design objects, it should release locks it acquired in the first phase, so that other tools can acquire locks on those objects. Some databases will, upon detecting that it has lost connection with a tool, assume that the tool's operation was aborted and release all locks that the tool had acquired, in order that the operation of other tools is not delayed.

Section 3.A.2: Traditional Workspaces

Stable designs are placed in the **public area** of the database. All updates to design data are encapsulated within work areas called **workspaces** [Gray 81]. A workspace is a region in the database which holds copies of design objects. Tools make changes only to objects in workspaces. These updates are tentative; a tool atomically **commits** changes to the public area when the desired state is achieved. The public area is thus used to hold designs which have achieved some level of correctness. 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 that workspace is the same as that offered by the public area. See Figure 15.

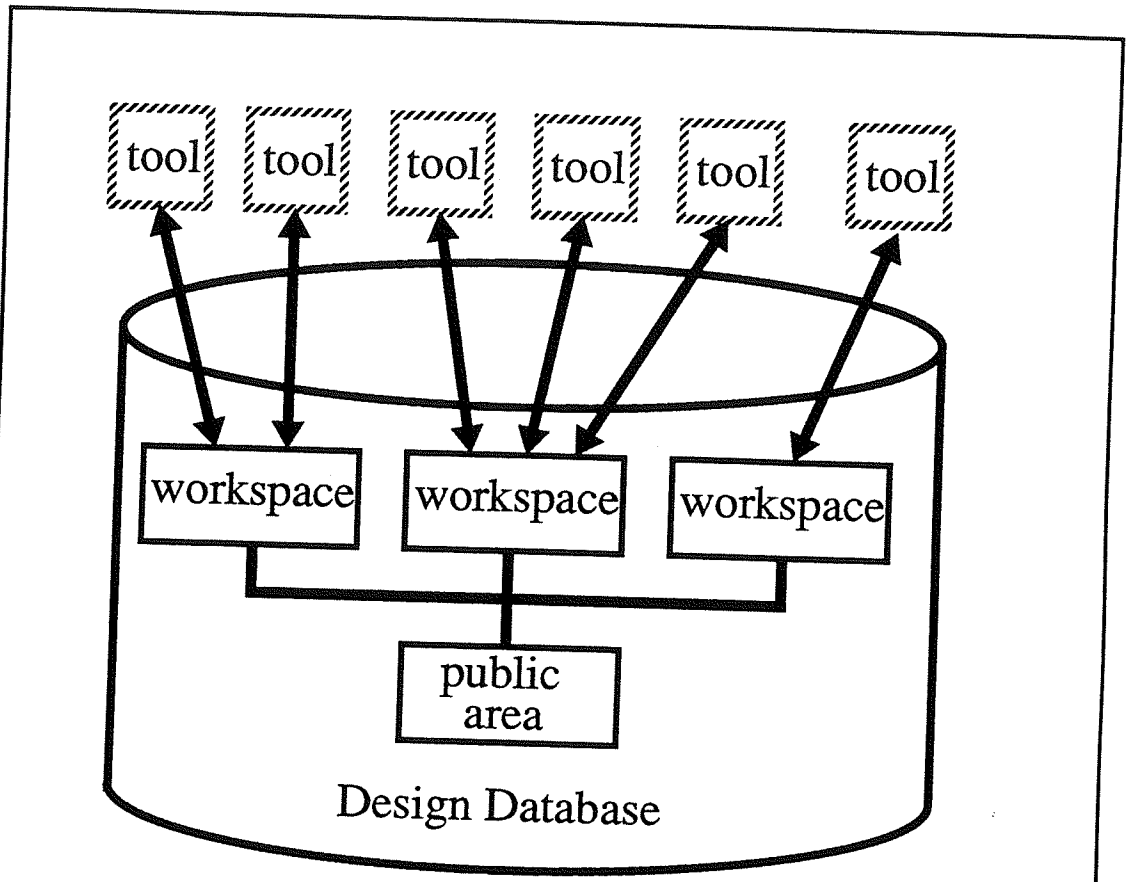

Figure 15: Workspaces.
Changes made by tools to design data in workspaces are tentative. When those changes are committed, they take effect in the public area, which holds stable designs.

Updates in Workspaces

Copies of design objects in the workspaces hold the tentative state of the objects. The update operations described in Part 2.F can be applied to the design objects in a workspace. A workspace offers a **view** of design objects, which is the collective state of the design objects. The view of the design objects offered by a workspace is the view of the objects in the public area modified by some delta; this delta is concatenation of all updates 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.

Commit and Abort

Let $V_W(t)$ and $V_P(t)$ represent the view offered by workspace W and the public area P at time t. Let u_i represent the i^{th} update to W, and $\Delta_W(t) = < u_1, u_2, ..., u_n >$ 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:

For all t, $V_W(t) = V_P(t) + \Delta_W(t)$. In other words, the state of objects in the workspace is the same as that in the public area excepting updates made to objects in the workspace.

If update u occurs at time t_u, then $\Delta_W(t_u) = \Delta_W(t_u - 1) + < u >$, which is to say that updates have a cumulative effect on the workspace.

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

Then for all t, $t_{prevCommit} \leq t < t_{commit}$, $V_P(t) = V_P(t_{prevCommit})$.

Furthermore, $V_W(t_{commit}) = V_P(t_{commit}) = V_P(t_{commit} - 1) + \Delta_W(t_{commit} - 1)$ and $\Delta_W(t_{commit}) = < >$.

In other words, updates in a workspace have no effect on the public area until the updates are committed, and all updates are applied atomically at commit time.

Suppose updates to W are aborted at time t_{abort}.

Then $V_W(t_{abort}) = V_P(t_{abort}) = V_P(t_{prevCommit})$ and $\Delta_W(t_{abort}) = < >$.

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

Section 3.A.3: Traditional Design Object Check-out & Check-in

Before a tool can read or update a design object in a workspace, it must **check-out** that design object into the workspace. Check-out is an association among tool, workspace, and design object. See Figure 16. Check-out may be made either for read or update access. A design object may be checked-out for update access by only one tool at any given time. Furthermore, checking-out an object for update access excludes read access by different tools. Thus in the traditional design environment the check-out of a design object D for update in a workspace W by tool T is an exclusive write-lock on D given to T which limits updates to D to occur only in W and only by tool T, and checking-out for read access is a shared read-lock.

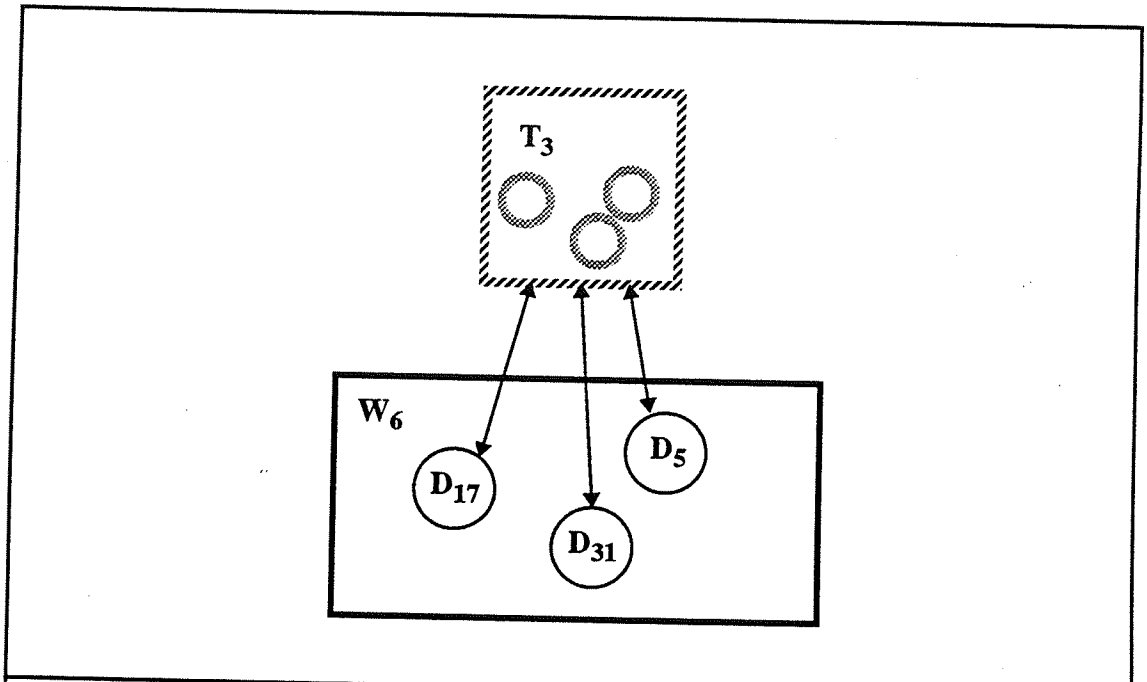

Figure 16: Design object check-out.
Tool T_3 checks-out design objects D_5, D_{17}, and D_{31} into workspace W_6, either for read or update, and caches copies of those objects internally.

Instead of checking out an existing version of a design object for update, a tool can create a new version of that object and then check-out the new version for update access. This new

version is initially a copy of what was previously the latest version. Updates are then applied to this new version. The tradeoffs of creating a new version versus undating an existing version are discussed in Section 4.B.7.

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. A tool must apply internal updates to the workspace or abort them before it checks-in design objects.

Section 3.A.4: Limitations of Traditional Tools

As described above, the design database offers the protocol of check-out and check-in which ensures that tools have exclusive access to design data for the duration of their operation. This is necessary because the tools have been written to assume that data in their **read set**, that is, those data upon which it has predicated its operation, are not changed by agents external to the tool. Allowing other tools to change those data might adversely affect the integrity of the tool's results.

Check-out uses an exclusive lock. So although it guarantees tools that their read sets will not be disturbed, concurrent operation of tools is still severely limited. The check-out/check-in protocol requires that tools which may need update access to design objects do not execute concurrently, which means that engineers must carefully plan which tools they will use when and on what design data.

The goal of concurrent design is to allow engineers to cooperate and their tools to share access to design data, so that the design in progress, rather than the tools in use, is the primary concern. For example, an engineer should feel free to invoke an editor and a simulator on a design, and would expect the simulator to act in a reasonable fashion (perhaps subject to user preferences) when he edits the design. Likewise, two engineers may wish to edit "different regions" of the same design object, and have informally agreed not to disturb each

other's efforts. Traditional design environments, because of their use of exclusive locking, prohibit both situations.

Integrated Toolsets

The usefulness of a set of CAD tools is limited by the degree to which they can work together. There have been attempts by tool vendors to build integrated toolsets [Henderson 89]. In an integrated toolset, each tool knows the data needs of every other tool; by intercommunicating the tools are able to stay consistent with one another. This requires that each tool stay aware of all tools which are running and inform them when relevant changes occur. For example, a toolset might consist of a layout editor, a schematic editor, and a simulator. See Figure 17.

There are several difficulties with the approach of building an integrated toolset:

- The complexity of intercommunication grows $O(n^2)$, where n is the number of tools.
- The approach taken in an integrated toolset requires that tools understand the semantics of other tools. But doing so means that if any tool changes, other tools must be modified to reflect that change.
- The addition of a new tool also requires modifications to existing tools.
- The communication mechanism is specific to a particular tool vendor. Thus the toolset can include only tools from that vendor. No single vendor can provide all the functionality that a designer may need, however.

Section 3.A.5: Haphazard Consistency Checking in Traditional Environment

In a traditional design environment, consistency checking is done in a manual and haphazard fashion. CAD Frameworks use timestamps on data files to detect constraints that may be invalid [Bushnell 86]. What is needed is the automatic invalidation of constraints when

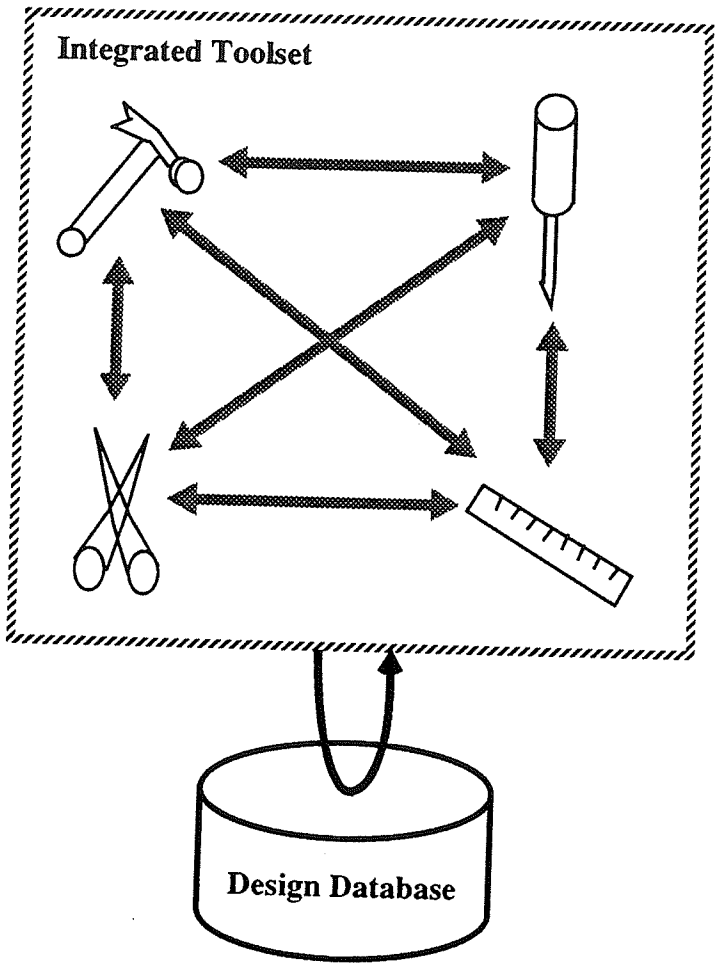

Figure 17: Integrated toolset.
Tools within the toolset share access to design objects and stay consistent by intercommunicating.

design objects upon which they depend have been updated. Requiring that all tools be aware of all constraints which have been defined by the schema, and trusting them to mark relevant constraints as invalid when updates occur, are unreasonable expectations of tools, however.

Part 3.B: Next-Generation Design Environment

This part of the chapter continues the discussion of Part 1.D. It explains those features of the next-generation design environment which support concurrent design and which are not offered by traditional tools and databases.

Section 3.B.1: No Exclusive Locking in Next-Generation Environment

As mentioned in Section 3.A.3, when a tool checks-out a design object for update, the database grants the tool an exclusive lock on the object. Thus the traditional design database controls access to design data in a manner analogous to concurrency control in a business transaction processing database. This should not be surprising, however, since most database technology in use today has its origin in business application. It is important not to make too much of this analogy, however, because the underlying paradigms are quite different.

In a business transaction processing system, transactions are short and are kept completely isolated from each other via exclusive locking or optimistic protocols [Papadimitriou 84]. Agents requesting the transactions are not allowed to assume that state will be retained across transactions. By contrast, in the design environment, transactions last much longer and are explicitly allowed to interact. Traditional techniques for handling concurrent access in a DBMS are not appropriate in a concurrent design environment, since a design database must permit transactions of arbitrary length which do not preclude access to data by many tools.

It is not feasible to place an exclusive lock on an entire design since many engineers work on overlapping aspects of it simultaneously. Even exclusive locking of 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 tools share updates to the same portion of a design. For example, two engineers might wish to share updates to the same portion. Or one engineer might want to run two

tools simultaneously (e.g., an editor and a simulator) on the same design data; tools must not be constrained to be invoked in a serial fashion.

Unlike the traditional design environment in which exclusive locking by the database ensures that tools can assume the locked data they access are static, in a concurrent design environment a number of tools may need to share updates to the same design objects. Without exclusive locks, there must be other mechanisms which permit tools to maintain views of the design consistent with the database.

Section 3.B.2: Information about Updates in Next-Generation Environment

Keeping a tool 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. Tools should not be expected to have knowledge of the semantics of other tools, however. Thus a central mechanism is needed which will notify a tool when data it has cached are changed by other tools. That mechanism is part of the Change Manager; it makes use of active data in the object store and is described fully in Chapter 4.

Section 3.B.3: Tools React to Changes in Next-Generation Environment

Even with a mechanism that guarantees tools that they are notified of changes to design data, each tool, in order for it to interact harmoniously with other tools, must react to these notifications in a proper fashion. This includes not only making its cache of design data consistent with the database, but also possibly undoing or making compensating changes to updates it had performed but not yet committed to the database. Exactly what a tool does depends upon the semantics of the tool and the design data. How notifications should, in general, be handled by a tool is discussed in Chapters 5 and 6.

Section 3.B.4: Use of Differential Update in Next-Generation Environment

Most tools, when they execute, make incremental rather than sweeping changes to the design. But if a tool submits its updates to the database as "the new state of the design" rather than "the differential changes applied to the design", 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 tools to take the form of a small rather than a large delta. A small delta can more easily be handled by a tool, and might be used by an intelligent tool to incrementally recompute the value of a computed slot. Tools in the next-generation design environment will update the database by submitting a list of differential updates $\Delta = < u_1, u_2, ..., u_n >$ in order to preserve knowledge of incremental updates.

Section 3.B.5: Next-Generation Environment Must Be Open-Ended

For the reasons discussed in Section 3.A.5, integrated toolsets are complex and of limited utility. It is not feasible for each tool to understand the semantics of changes made by all other tools; indeed, since new tools will be added, it cannot be anticipated what their functionality will be. Instead, tools must share an open-ended environment which can be extended to accommodate new tools without necessitating change to existing tools or other parts of the design environment. Furthermore it must accommodate sets of tools which are tightly coupled (such as two tools sharing updates to the same design objects), as well as those which are loosely coupled (such as tools under the control of different engineers working on different aspects of the same design).

Section 3.B.6: Support for Cooperation in Next-Generation Environment

Engineers use tools in separate workspaces when the design objects checked-out into those workspaces are unrelated, or when integration of design objects into a parent design is being deferred. At other times, when engineers want to work on very closely related parts of the design, any partitioning may seem artificial and may impose an unacceptable overhead. In this case, they will use tools that access the same design objects in the same workspace. See Figure 18. When tools share access to a design, the tools' views of the design will be kept synchronized with that of the workspace through notifications.

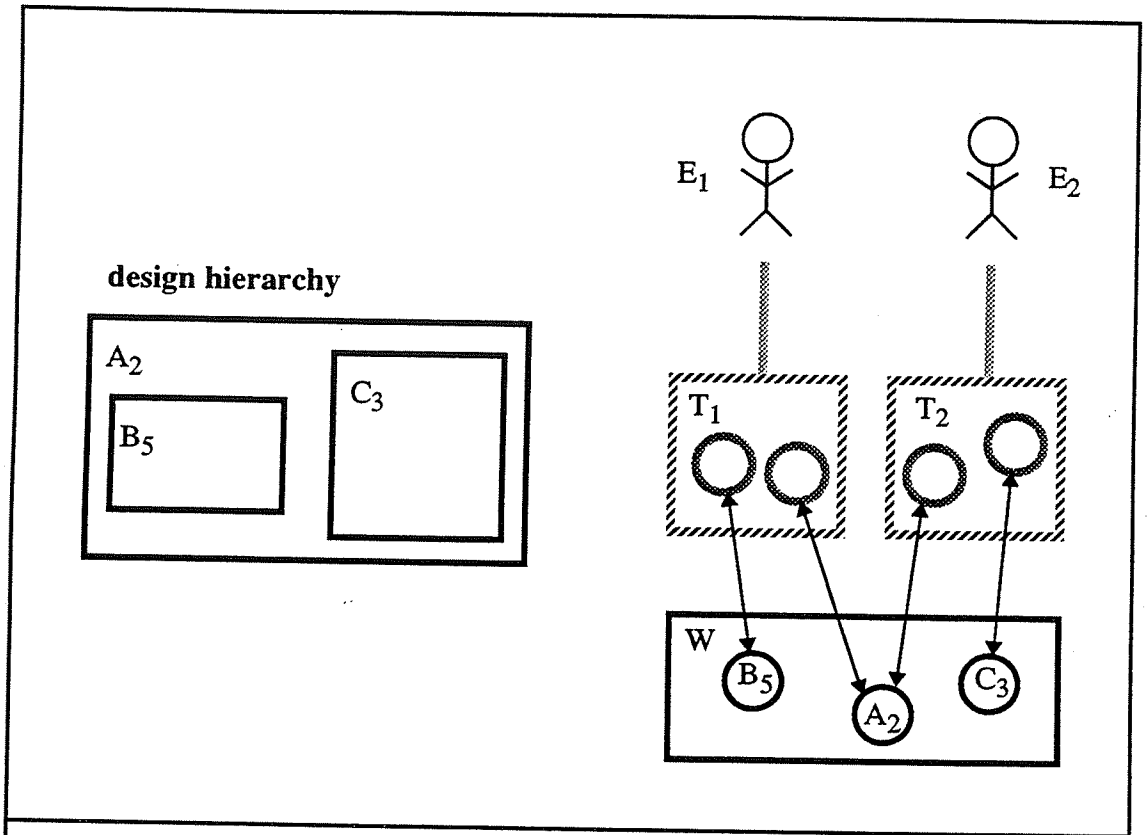

Figure 18: Close interaction among engineers.
In this figure, engineer E_1 uses tool T_1 to update B_5 in workspace W, and engineer E_2 uses tool T_2 to update C_3 in W. Both E_1 and E_2 update A_2 in W, because changes to B_5 and C_3, respectively, may require parallel adjustments to A_2.

It is important to note that using tools constructively in this fashion depends upon informal communication among the agents using them. This concept of cooperation among agents is absent in the traditional design environment but is an important aspect of the next generation design environment which will support concurrent design.

Section 3.B.7: Next-Generation Environment Offers Workspace Hierarchy

The notion of workspace, as presented in Section 3.A.2, can be generalized to a hierarchy [Moss 85]. See Figure 19. At the top of the hierarchy is the **root workspace** W_{root}. Every

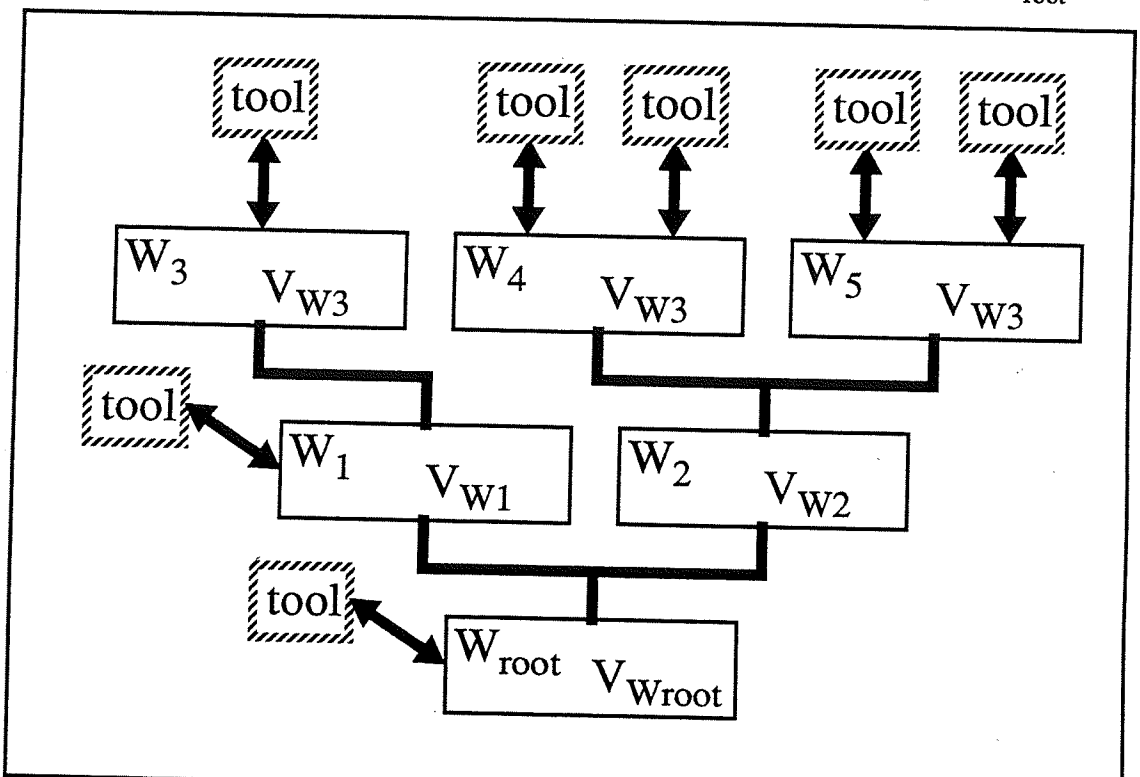

Figure 19: Workspace hierarchy.
Tools check-out design objects in workspaces. Tool updates are encapsulated within a workspace, and propagate to the superior workspace at commit time. An inferior workspace offers a view of design objects which is that of the superior workspace modified by the updates which have been applied to the inferior workspace.

workspace W except W_{root} has a superior workspace Superior(W). The root workspace holds archived designs, libraries of components, and fully validated designs. "Super" workspaces, that is, those closer to W_{root}, hold design 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 supervisor may create sub-workspaces in order to separate unrelated projects or to create a work area with consistency requirements (described in the next section) less stringent than those of the root workspace. An engineer may create an inferior workspace in order to encapsulate tentative or experimental updates, or narrow his focus to some subset of design objects.

Let W_1 and W_2 be workspaces. We define the $\leq$ relation between workspaces to be the reflexive and transitive closure of the "inferior-of" relation as follows:

$W_1 \leq W_1$

$W_1 \leq W_2$ implies $W_1 \leq \text{Superior}(W_2)$

If $W_1 \leq W_2$ and $W_1 \neq W_2$, then W_1 is said to be an **sub-workspace** of W_2, and W_2 is said to be a **super-workspace** of W_1.

The workspace hierarchy has invariants and semantics of commit and abort that are completely analogous with those presented in Section 3.A.2. Given workspace $W \neq W_{root}$:

For all t, $V_W(t) = V_{Superior(W)}(t) + \Delta_W(t)$.

If update u occurs at time t_u, then $\Delta_W(t_u) = \Delta_W(t_u - 1) + < u >$.

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

Then for all t, $t_{prevCommit} \leq t < t_{commit}$, $V_{Superior(W)}(t) = V_P(t_{prevCommit})$.
Furthermore, $V_W(t_{commit}) = V_{Superior(W)}(t_{commit}) = V_{Superior(W)}(t_{commit} - 1) + \Delta_W(t_{commit} - 1)$ and $\Delta_W(t_{commit}) = <>$ (the empty list).

Suppose updates to W are aborted at time t_{abort}.

Then $V_W(t_{abort}) = V_{Superior(W)}(t_{abort}) = V_{Superior(W)}(t_{prevCommit})$ and $\Delta_W(t_{abort}) = <>$.

Along with the workspace hierarchy come generalized forms of check-out and check-in. See Figure 20. A design element can be updated only in the workspace in which it is cur-

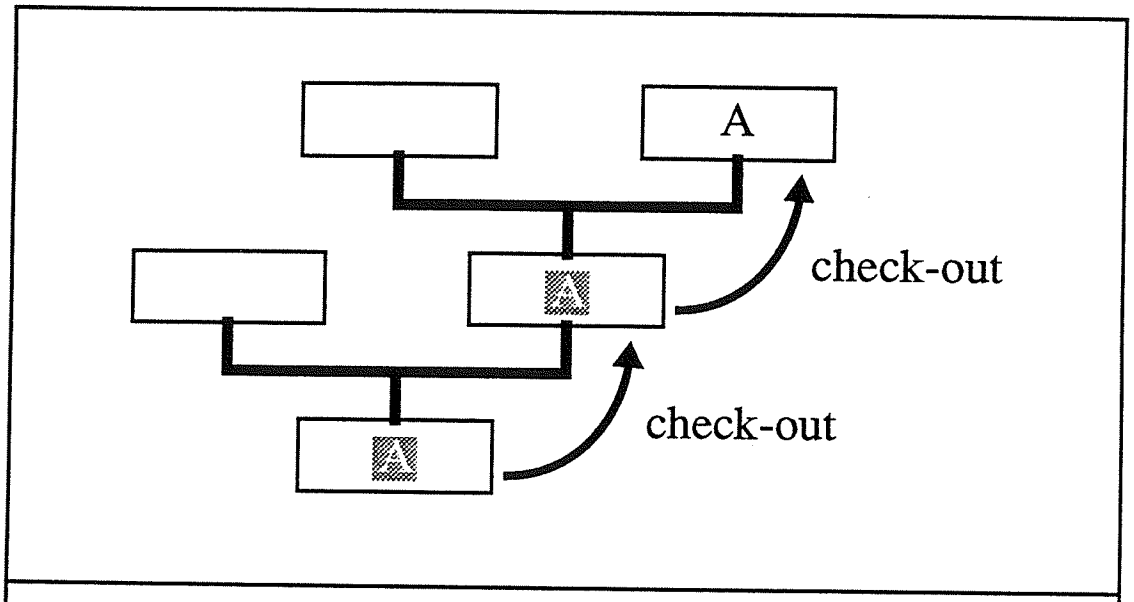

Figure 20: Generalized check-out.
A design object can be checked-out into a sub-workspace in order to make tentative updates which will later be committed to the superior workspace.

rently checked-out for update. The rules of check-out and check-in are described fully in Chapter 4.

In a DDMS which offers two levels of workspace—public and private—the actions of check-out and check-in of a design object strictly alternate. What is needed is a dynamic hierarchy of workspaces for agents and their tools which permits a subworkspace to be created at any time. In that subworkspace a subset of design 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 committed atomically to the superior workspace. The Change Manager offers a programmatic interface to manipulate a hierarchy of workspaces; the interface will be presented in Chapter 4.

Section 3.B.8: Constraint Requirements in Next-Generation Environment

The object model described in Chapter 2 mentions constraints, which are types of computed slots. Constraints among design data must at some point in the design be ascertained to be valid. The following are example of constraints:

- The bounding box of a design element may be constrained to be small enough so as not to overlap neighboring modules.
- Two design elements are alternate views of the same component and should at some point in the design become equivalent.
- The implementation of a design element matches its specification.

One restriction which is intended to limit the propagation of potentially incorrect changes 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 the DDMS. In traditional design environments this task is performed manually. The manual method is error prone, however: an engineer may forget to invoke tools to check consistency, or may be tempted to give intuitive (and often incorrect) approval of the updates performed.

The Engineering Information System (EIS) project [Hall 88] incorporates a rulebase, and a corresponding mechanism to interpret that rulebase, into the design environment for the purpose of automatically running validation tools on modified designs. The Change Manager offers **constraint requirements** in workspaces as part of the next-generation design environment. A constraint requirement is an attachment to a workspace that names a constraint in design objects which must be known to be valid within a tool cache (or in an in-

ferior workspace) before the tool (or inferior workspace) can commit to that workspace. Constraint requirements are inherited by super-workspaces. See Figure 21.

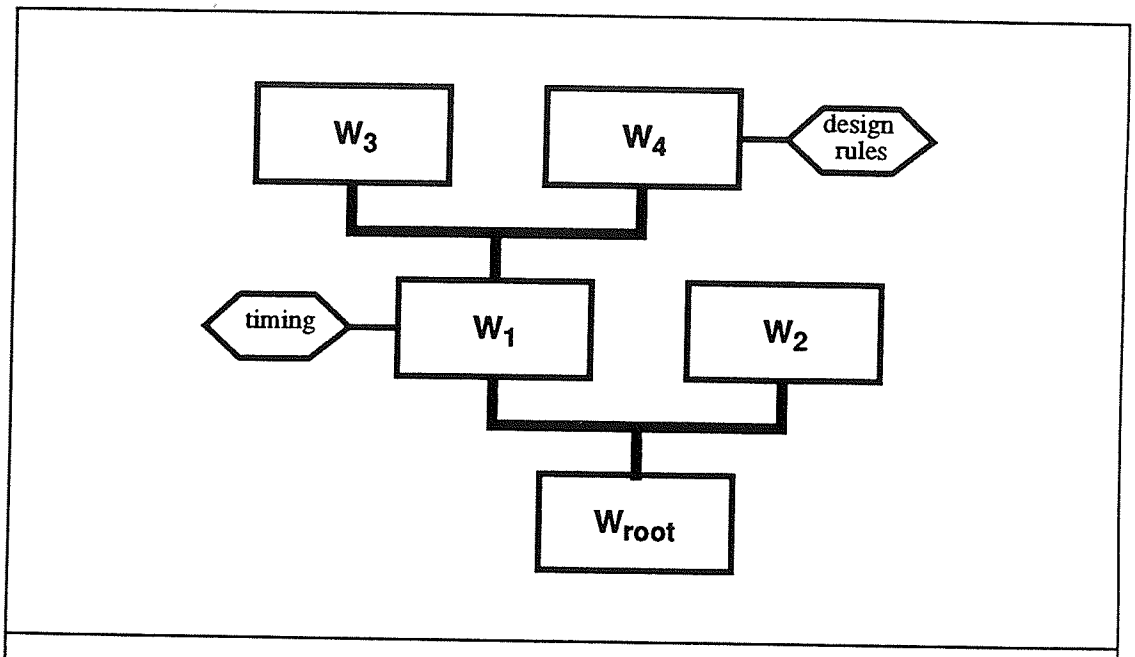

Figure 21: Constraint requirements.
In this figure, workspace W_4 guarantees that design rules are met by objects in that workspace, and requires that updates to W_4 preserve the validity of design rules. Workspace W_1 inherits the constraint requirement of design rules from W_4, and additionally requires timing verification. The root workspace W_{root} requires both that design rules are met and that timing has been validated.

A supervisor can use constraint requirements to enforce some subset of constraints in certain workspaces in order to guarantee a known degree of consistency within that workspace. The supervisor can assign different constraint requirements to different workspaces depending upon the degree of correctness required. A public workspace might have strict requirements, whereas an experimental workspace might have none, for example.

A constraint requirement does not specify how a constraint is to be validated, nor when. It is merely a restriction on committing changes to a workspace which is based upon the status of constraints. Other mechanisms would be needed to control when to fire consistency checkers.

Section 3.B.9: Conflict Logging in Next-Generation Environment

Even when engineers are benevolent and attempt to cooperate, and the tools under their control are operating correctly, there may be times when one engineer will make a change to a design object that another engineer cannot understand, cannot adapt to, or considers an error, and is therefore unacceptable. This is knows as a **conflict**. Conflicts 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 design object.

When a conflict occurs, an agent or the agent's tool may wish to register disapproval of the update in an effort to obtain corrective action or an explanation. That is done with a **conflict record** that references the offending and offended agents, the update which caused the conflict, and the tool which performed the update. The engineers normally will try to resolve the conflict between themselves. If they cannot, then resolution of the conflict is the responsibility of the supervisor. A **conflict resolution** is a record that some action was taken on behalf of the conflict. Such action might be a retraction of the offending update or an overriding approval by a supervisor. A record of conflicts and their resolutions are kept for each workspace both in order to provide a history and to enable a supervisor to browse unresolved conflicts.

In order to limit the propagation of conflicting updates, the design environment should prohibit a workspace from committing to its superior workspace if it contains unresolved conflicts. Support for conflict logging requires that change notifications sent to tools include the identity of the engineer and tool responsible for the change. The traditional design environment offers no support for conflicts.

Section 3.B.10: Design Status in Next-Generation Environment

As mentioned in Section 1.C.2, in the next-generation design environment a portion of the internal state of a design database is an extension of the design data, and is referred to as **design status**. Here are some examples of design status:

- which versions of a design element exist
- the hierarchy of workspaces
- which agents are running what tools
- which tool has checked-out what design object in what workspace

Note that information about tools which are currently running is part of the design status. In order to acquire this information, CAD tools are required to register (unregister) themselves with the DDMS when they begin (respectively, end) operation.

Access to the design status can assist an engineer in coordinating design activities with those of other engineers. For example, if engineer E_1 is updating design object Z, the knowledge that another engineer E_2 is also updating Z will alert E_1 to potential conflicts, and may catalyze communication between E_1 and E_2. Or an engineer may want to check what constraint requirements have been attached to the workspace to which the design will eventually be committed. Design status is useful for a supervisor, too, who wants to monitor shared access to designs, conflicts, or the creation of new versions, for example, in order to monitor progress of the design [Roussopoulos 91].

Traditional design databases do not make design status accessible to tools. The next-generation design environment must not only provide access to the design status, but make it **active**, that is, allow tools the ability to track changes in design status.

Part 3.C: The Change Manager

Section 3.C.1: Definition

The two preceding parts of this chapter have described in what ways traditional databases and CAD tools are inadequate for an environment of concurrent design, and have proposed enhancements that overcome these deficiencies. The remainder of this thesis builds upon the object model discussed in Chapter 2 and presents the **Change Manager**, which is a collection of software modules, distributed among the DDMS and tools, which collectively provide the framework for this enhanced capability.

The Change Manager is a software layer that sits above the repository of design data but below the applications which manipulate those data, thereby acting as an intermediary between the application code in the tools and the data store in the DDMS. Operationally, application code within a tool invokes libraries of the Change Manager which have been linked with the tool—the Tool Change Manager or TCM—the TCM interacts with the Change Manager in the DDMS—the DDMS Change Manager or DCM—and the DCM invokes functionality of the data store. Procedure calls are synchronous, that is, the caller is blocked until the procedure completes. Because of the use of triggers to achieve active data in the data store, notifications are passed asynchronously from the data store through the Change Manager to the application. See Figure 22.

Section 3.C.2: Requirements of the Change Manager

In order to provide services which are needed in the next-generation design environment which supports concurrent design, the Change Manager must meet the following requirements:

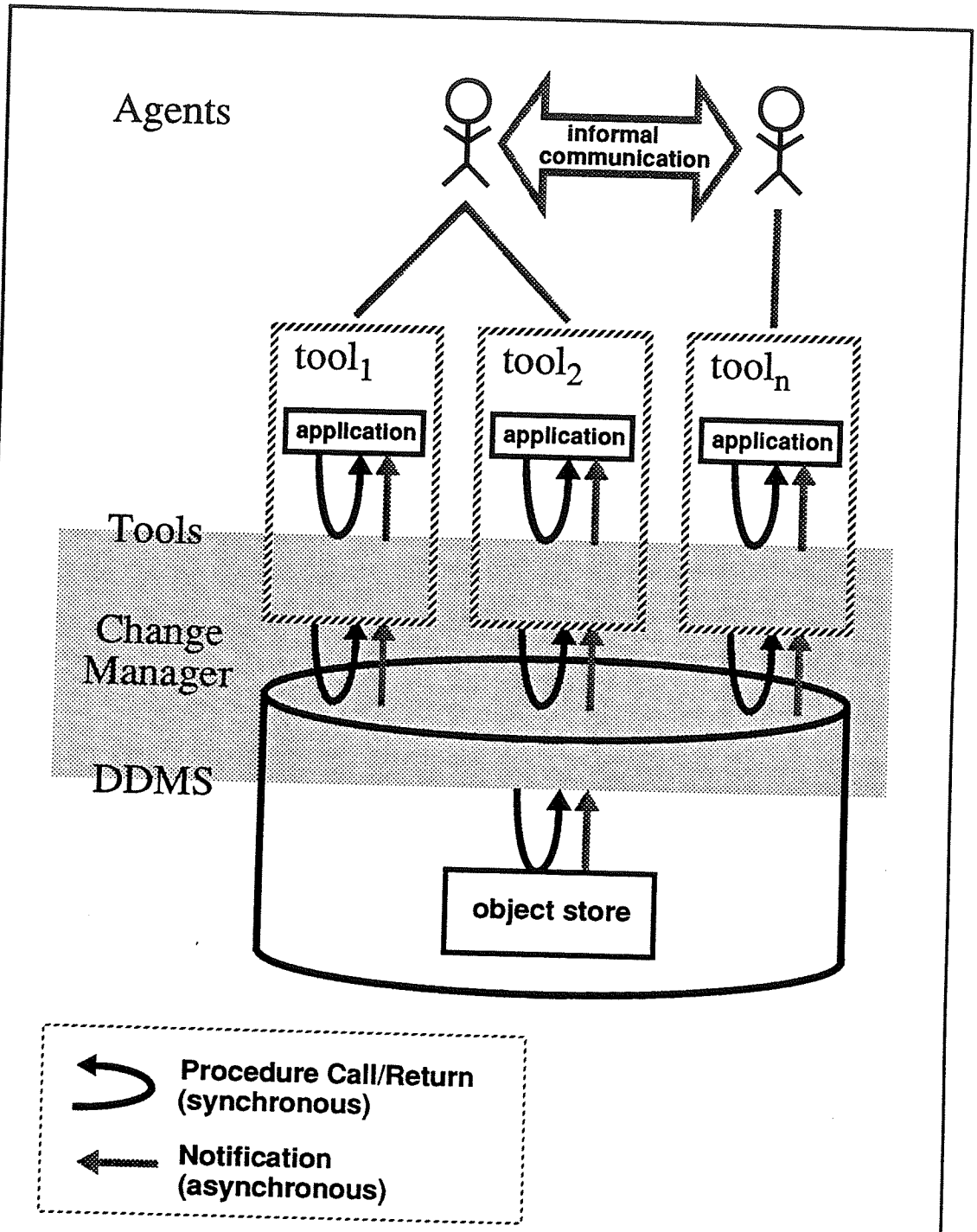

Figure 22: Use of the Change Manager in the design environment.
The Change Manager offers services to CAD applications and augments the functionality of an object-oriented data store in order to make them suitable for use in a concurrent design environment.

- support for varying degrees of cooperation

  Facilities provided by the Change Manager must accommodate sets of tools which are tightly coupled (such as a schematic editor and a simulator which are being used simultaneously by one designer) as well as those which are loosely coupled (such as tools under the control of different engineers working on different aspects of the same design). Thus it cannot make use of exclusive locking in order to ensure consistency of design data.

- use of notifications rather than exclusive locking

  The DCM tracks updates to shared data. Unlike a traditional database in which the guarantee given to an application is that it has exclusive access to design data, the DCM instead guarantees only that a tool will receive asynchronous notifications of changes to data it is accessing. The tool can use these notifications to maintain a view consistent with the database. Cooperating engineers communicate informally; the DCM formalizes asynchronously communication between the DDMS and tools.

- open-ended

  The Change Manager is independent of the semantics of particular applications within tools, so that new applications can be added to the environment without necessitating modification to the Change Manage. Thus, in order for the Change Manager to track changes to design data in which an application is interested, each application must inform the TCM of the set of updates in which it is interested. This requires that the TCM offer a programmatic interface to applications with which they can specify that set of updates.

- workspace hierarchy

  The DCM offers a hierarchy of workspaces and associated check-out and check-in protocols with which to encapsulate tentative changes to design data.

- constraint requirements

  The DCM enforces consistency constraints which have been attached to workspaces.

- conflict mechanism

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

- design status

  The Change Manager gives tools access to the design status.

- automatic tool cache consistency

  Tools cache design objects which they are accessing. A cache may grow stale, however, when another tool updates those design objects. The TCM processes update notifications from the DCM and ensures that the cache stays consistent with the DDMS in the face of updates by other tools.

- automatic voiding of constraints

  It is unreasonable to assume that applications will understand the impact of updates they make on all constraints in the design. The set of constraints may grow over time, for example, as the data model evolves. For this reason, the TCM is responsible for voiding constraints whose validity may have been disturbed by updates. This is discussed fully in Chapter 5.

- efficiency

  When modules of the TCM which are linked within a tool are invoked by an application, the CPU of the workstation running the tool is used. Thus the TCM must be reasonably efficient and not significantly degrade the performance of individual tools. The DCM manipulates and controls access to data in workspaces, but the manipulations of design 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.

- programmatic interface to applications offered

  The TCM offers a programmatic interface and associated protocol with which applications can create, destroy, commit, and abort workspaces, check-out and check-in design objects, and access and update design data.

Section 3.C.3: Architecture and Operation

Remaining chapters of this thesis present in detail the architecture and operation of the DCM and TCM, explain what capability they add to the DDMS and tools, respectively, and show how that capability provides what is needed in the next-generation design environment (as described in Part 3.B) and meets the requirements listed above.

Chapter 4

Change Manager Support for the DDMS

The Design Data Management System (DDMS) is the design database of the next-generation design environment, as described in Part 3.B, which will support concurrent design. It follows the paradigm of a server, whose function is to await and service requests from clients, in this case CAD tools. The DDMS is unlike a server, however, in that servicing a request from one tool may cause asynchronous notifications to be sent to other tools. This chapter presents the architecture of the DDMS, describes what functionality the DDMS Change Manager (DCM) adds to an object-oriented data store in order to overcome the weaknesses discussed in Chapter 3, presents the programmatic interface between CAD tools and the DDMS, and summarizes the invariants maintained by the DCM.

Part 4.A: Architecture of the DDMS

Section 4.A.1: Introduction

The Design Data Management System consists of an object store plus the DCM. The object store provides persistent storage of the schema, annotated objects (explained in Section 4.B.5 below) which store design data, and access permissions to those objects. The DCM consists of seven modules: DDMS clock, tool registry, workspace manager, data access module, update monitor, design status monitor, and conflict logger. See Figure 23.

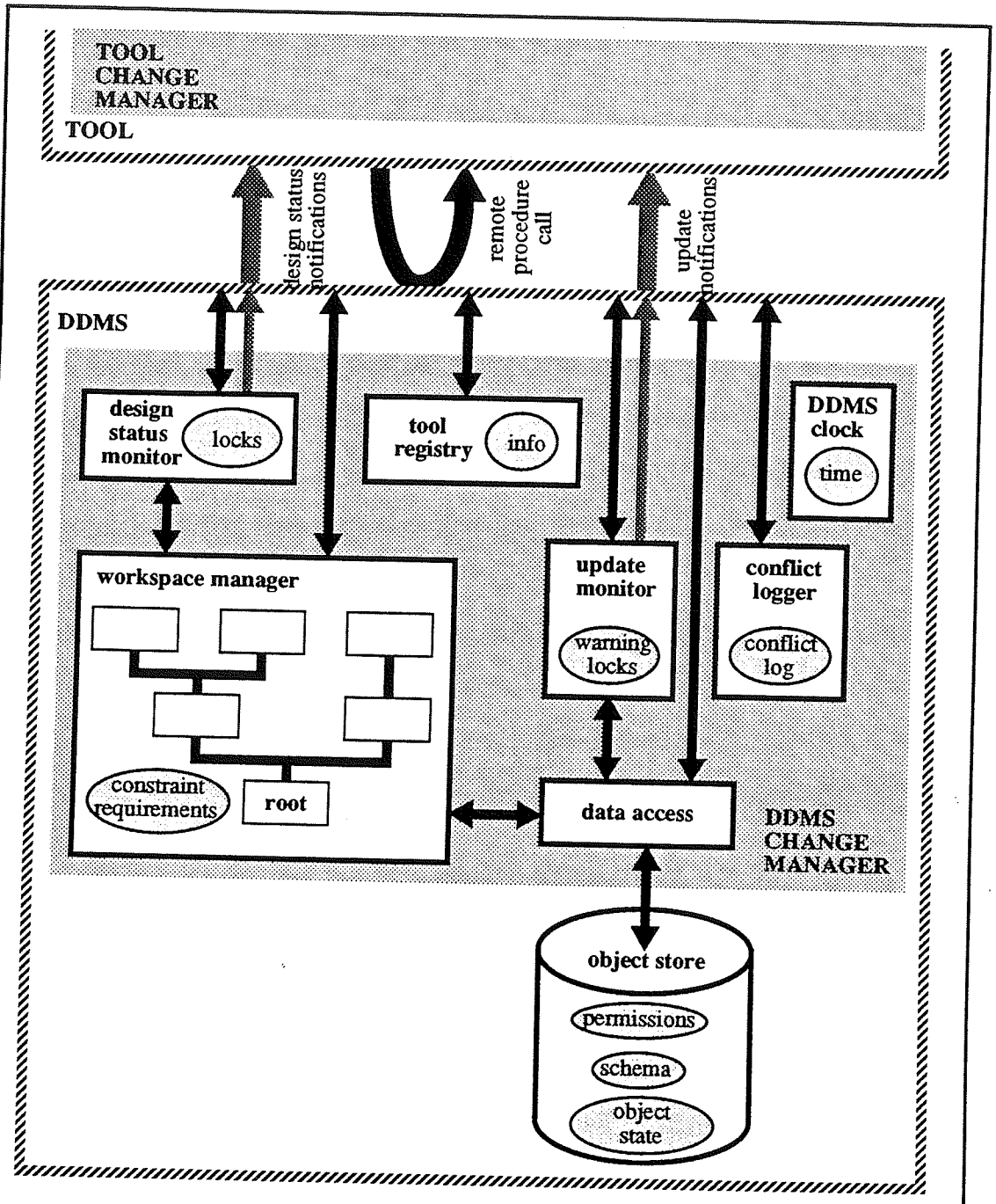

Figure 23: Architecture of the Design Data Management System.
The DDMS consists of the object store plus the DDMS Change Manager. The DDMS Change Manager is a layer between CAD tools and the object store; it consists of seven modules: DDMS clock, tool registry, workspace manager, data access module, update monitor, design status monitor, and conflict logger.

These seven modules work together and with the help of the object store offer a collection of services to CAD tools.

In the next-generation design environment, agents will run tools locally on their own workstations. Tools are run asynchronously with respect to one another. The tools will communicate to the DDMS through the use of inter-process communication (IPC). The latency of IPC is high compared to communication within a workstation. So the choice of what functionality to assign to the DDMS has been motivated by the need to reduce the frequency of interaction of tools with the DDMS. In order to accomplish this, the programmatic interfaces presented in this chapter specify a granule of operation at the level of *design object*, rather than at the level of *slots*.

Section 4.A.2: DDMS Clock

The DCM maintains an integer-valued clock in the DDMS. The clock is incremented (at least) whenever the DDMS processes each request from any tool. If a request contains many subrequests, such as when a tool commits a collection of updates to the DDMS, the clock is incremented many times, once per subrequest. If the DDMS receives requests from multiple tools at the same time, the requests are arbitrarily ordered and placed in a queue to be processed as soon as possible.

The timestamp of some operations, such as a tool committing a batch of updates, is remembered by the DCM for later use. Other timestamps, such as the time at which the slot of an object changed value, are stored in the object store. Because the clock is incremented after each request, timestamps are unique. If the DDMS is distributed, then methods for event ordering can be used to ensure uniqueness of timestamps [Lamport 78].

Part 4.B: Functionality of the DDMS Change Manager

This part of the chapter describes each service that the DCM offers to CAD tools, presents the programmatic interface which a tool uses to access the service, and explains how modules within the DCM operate in order to provide that service.

Section 4.B.1: Tool Registration

Some of the information maintained by the DCM, such as which objects have been checked out, is associated with a particular instance of a tool. Thus each instance of a tool has its own identity. That identity is established when a tool registers itself, and is removed when a tool unregisters.

Programmatic Interface

RegisterTool(AgentName, ToolName)
returns ToolID

When a tool begins execution, it must register itself with the DDMS. The DCM records what tool is running and what agent is operating the tool, and returns a tool ID which uniquely identifies that instance of the tool. The name of the tool and name of the agent can be accessed as part of the design status, as described in Section 4.B.11. The identity of the agent using the tool determines what access permissions are granted to the tool. The tool ID returned by the DCM is used in subsequent requests to the DDMS to identify the tool making the request.

UnregisterTool(ToolID)

When a tool terminates, it must unregister itself. The DCM invalidates the tool ID and removes the name of the tool and agent from the list of currently executing tools in the design status. If a tool has a workspace selected (defined in Section 4.B.3), the tool must unselect the workspace before it can unregister itself.

Section 4.B.2: Creating and Destroying Workspaces

As explained in Section 3.B.7, a hierarchy of workspaces is useful in a concurrent design environment. The DDMS offers tools the ability to create and destroy workspaces and to determine what workspaces exist. (It might be useful to limit creation and destruction of workspaces to supervisors' tools. Exactly what permissions are needed is a policy question, and is beyond the scope of this thesis, however.)

Programmatic Interface

CreateWorkspace(ToolID, SuperiorWorkspaceID [, **set** InferiorWorkspaceID])
returns WorkspaceID

CreateWorkspace creates a new inferior workspace of a specified superior workspace and returns the unique ID of that new workspace. A set of IDs of workspaces which are inferiors of the specified superior workspace can optionally be supplied; doing so will make them inferiors of the new workspace. See Figure 24.

Initially the view of design objects in the new workspace is inherited (and is thus the same) as the view in its superior, and the set of constraint requirements (explained in Section 4.B.4) is the union of those of its inferior workspaces (or the empty set ∅ if no initial inferior workspaces were specified).

DestroyWorkspace(ToolID, WorkspaceID)

DestroyWorkspace 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 tool has W selected (defined in Section 4.B.3)
3. there are no uncommitted updates in W

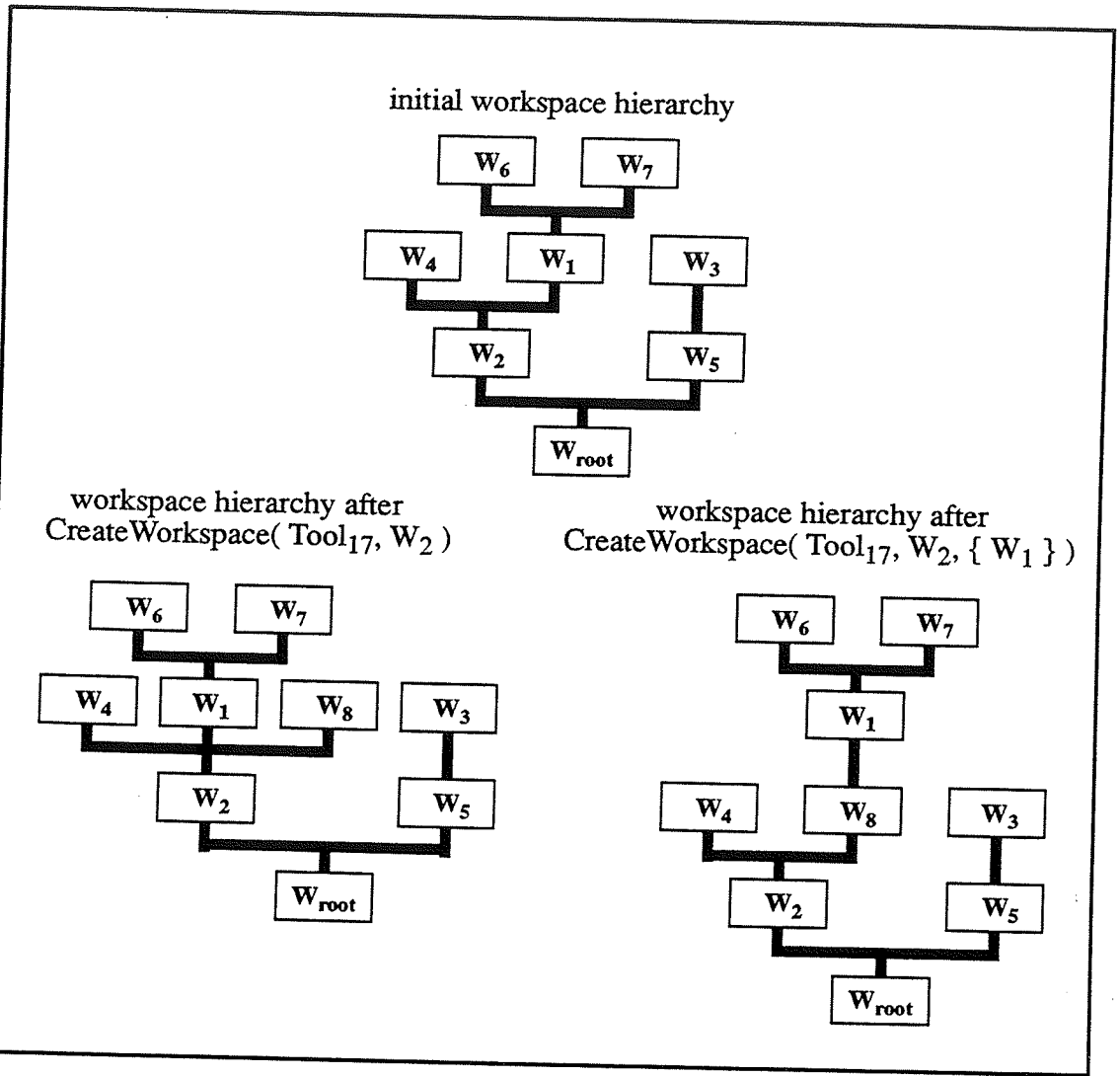

Figure 24: Creating a workspace.
At the top is an example workspace hierarchy. On the left, workspace W_8 is created initially with no inferior workspaces. On the right, W_8 is instead created with initial inferior workspace W_1.

If there are uncommitted changes in W, W must first be committed or aborted before being destroyed. When a workspace is destroyed, its inferior workspaces become inferiors of its superior workspace. See Figure 25.

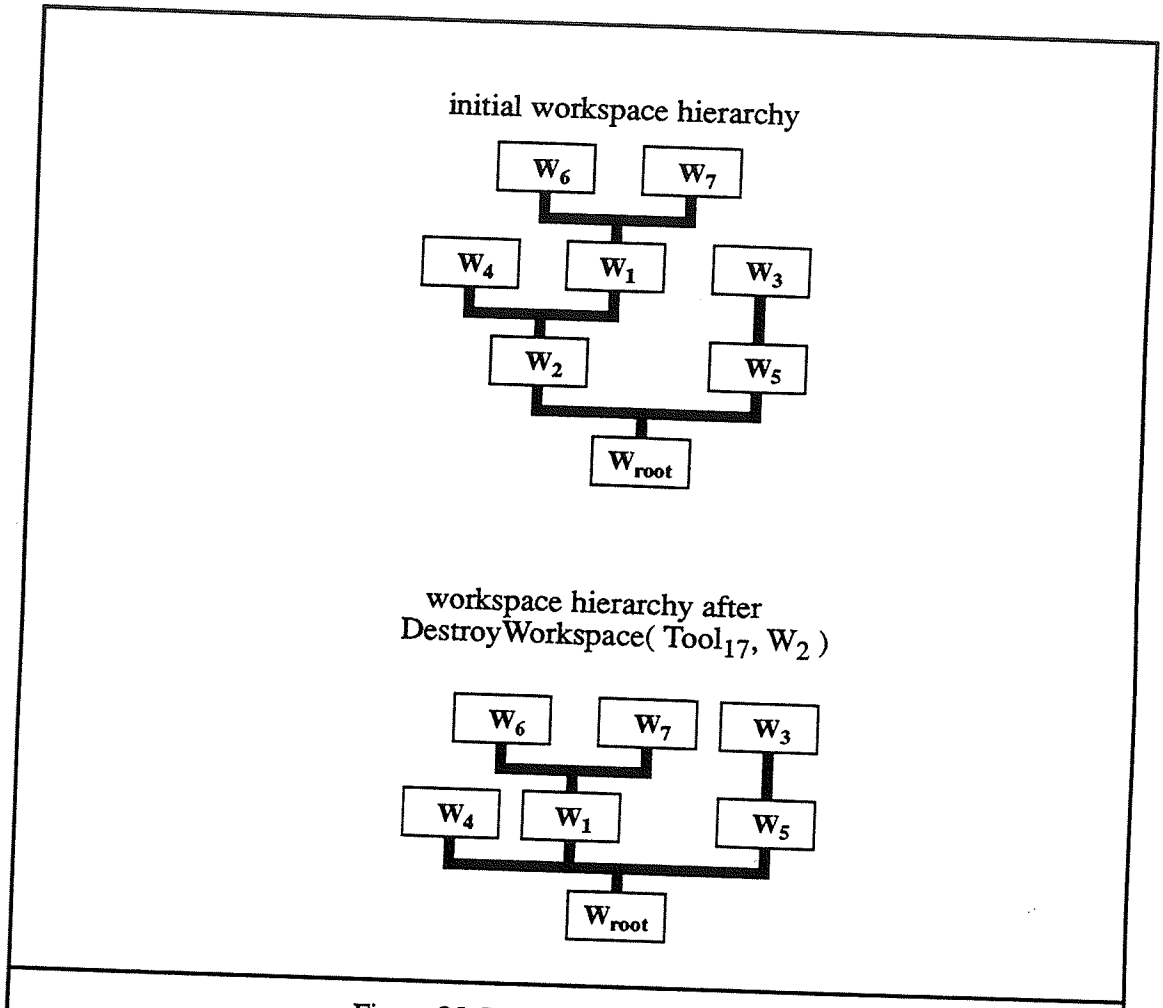

Figure 25: Destroying a workspace.
The superior workspace inherits the workspace's inferior workspaces when the workspace is destroyed.

GetWorkspaceInferiors(ToolID, WorkspaceID)
returns set WorkspaceID

A tool can determine the inferiors of a given workspace by calling GetWorkspaceInferiors. A tool can traverse the hierarchy of workspaces by recursive use of GetWorkspaceInferiors, starting with W_{root}.

Section 4.B.3: Workspace Selection

When a tool accesses and manipulates design data, it does so within a particular workspace. The choice of workspace depends upon the degree of cooperation and interaction desired with other agents and their tools. When two tools share a workspace they can work together closely and share updates to design objects. The choice of workspace also depends upon how stable a view of design objects is needed by the tool, as discussed in Section 3.B.7.

The tool must inform the DDMS in which workspace it needs to operate; this is called **workspace selection**. After a workspace is selected, the operations of check-out and check-in are performed with respect to the workspace selected. If a tool needs to work in a different workspace, it must unselect the selected workspace and select the other workspace.

Programmatic Interface

SelectWorkspace(ToolID, WorkspaceID)

SelectWorkspace informs the DDMS that a tool wants to access design data within a particular workspace. A tool can have at most one workspace selected at any time.

UnselectWorkspace(ToolID)

UnselectWorkspace informs the DDMS that a tool has finished working in a workspace. With no workspace selected, a tool cannot check-out design objects. Before a tool can unselect a workspace, it must check-in any design objects that it had checked-out.

Section 4.B.4: Constraint Requirements

As discussed in Section 3.B.8, constraint requirements, which are attached to a workspace, specify that a specified subset of constraints must be met both before and after any set of changes is applied to objects in that workspace. A supervisor can use this facility to ensure that designs meet certain standards before they are admitted to super workspaces that are

more publicly accessible, as mentioned in Section 4.B.5. Consistency requirements provide assurance to any engineer who uses a workspace that design objects in that workspace conform to a certain level of validation.

The DCM enforces constraint requirements by rejecting a set of updates from a tool to the workspace if one or more of those constraints is not valid; it also prevents an inferior workspace from committing to that workspace if in the inferior workspace one or more of those constraints is not valid. The DCM offers an interface to add or remove constraint requirements from a workspace.

Workspaces inherit constraint requirements from inferior workspaces. Thus the set of constraint requirements 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 designs which have achieved the highest level of validation, has a large number of constraint requirements.

Tools will normally operate in workspaces which have few constraint requirements, since the use of a tool usually involves updates made interactively by a design engineer during which no particular degree of consistency of the design is expected to have been achieved.

Programmatic Interface

AddConstraintRequirement(ToolID, WorkspaceID, ObjectType, Slot)

AddConstraintRequirement adds a constraint requirement to a specified workspace. The constraint is specified as a slot of a particular object type, as defined by the schema; the slot must be Boolean-valued. A constraint requirement 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.

RemoveConstraintRequirement(ToolID, WorkspaceID, ObjectType, Slot)

RemoveConstraintRequirement removes a constraint requirement from a specified workspace and all sub-workspaces. Subsequent updates to those workspaces are accepted by the DDMS even if the value of that slot is **false** or **void**.

GetConstraintRequirements(ToolID, WorkspaceID)
returns set Constraint

A tool can query the DDMS to determine what constraint requirements have been assigned to a workspace by using GetConstraintRequirements.

Section 4.B.5: Committing and Aborting Workspaces

After designs in a workspace have achieved desired states of completion, it is useful to make them more public or to move them to a workspace used for integration with the efforts of other engineers. This is accomplished by committing the workspace to its superior workspace. As explained in Section 3.B.7, immediately after the commit, design objects in the committed workspace and its superior have the same state.

One way to implement the commit operation would be for the DDMS to copy any design objects which have changed since the last commit into the superior workspace. But this would destroy knowledge of incremental updates, as explained in Section 3.B.4. In order not to lose this knowledge, the DDMS maintains for each workspace W in the workspace hierarchy the update delta between Superior(W) and W; when W commits, that delta is applied to Superior(W). That last sentence is not quite true: the DDMS doesn't maintain the update delta, but maintains information in **annotated objects** which it uses to easily compute the delta.

Annotated Objects

The object store provides persistence for design objects, that is, it stores the values of their slots. So that the DCM can remember what updates have been applied to each workspace, the object store holds additional information for each workspace about each design object that has been altered in the workspace. The result is an **annotated object**. The information contained in design objects and in each type of slot of design objects is described in Table 1.

When a workspace W is committed, the DCM scans the objects in the workspace and uses the annotations 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 annotated 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 and applied to Superior(W). Instead, annotations of design objects in W and in all sub-workspaces are merely discarded. A workspace is aborted only if the updates that have been applied to design objects in the workspace are to be undone.

The timestamps in the annotated objects assume the value T of the DDMS clock at the time when the DDMS processes an update request by a tool. There is enough information in annotated objects without the timestamps to enable the DCM to infer the update delta. Timestamps are used for another reason: by comparing the timestamp of a computed slot to the timestamps of the slots from which it is computed, it is possible for the dependency locator in a CAD tool (presented in Section 5.B.7) to determine which slots were changed and caused a computed slot to be made void; this can potentially save a tool a great deal of effort in recomputing the computed slot. Timestamps are also used to ensure that a tool keeps a consistent view of design objects in its cache; this use is explained in Section 4.B.6.

Table 1: Annotated Design Objects

datum	annotation	value	meaning
design object	existence status	created in superior; unchanged	The design object exists in the superior workspace, and has not been destroyed in this workspace.
		created in superior; destroyed	The design object exists in the superior workspace, but has been destroyed in this workspace.
		destroyed in superior; unchanged	The design object formerly existed in the superior workspace, was destroyed in that workspace, and has not been undestroyed in this workspace.
		destroyed in superior; undestroyed	The design object formerly existed in the superior workspace, was destroyed in that workspace, but has been undestroyed in this workspace.
		not in superior; created	The design object was created in this workspace.
		not in superior; destroyed	The design object was created then destroyed in this workspace.
	value status	same	No slot of object has been updated in this workspace.
		different	Some slot of object has been updated in this workspace.
	timestamp	some DCM time	This is the time at which slots of the object were most recently updated.

Table 1: Annotated Design Objects

datum	annotation	value	meaning
primitive slot	value	some value of the appropriate type	The value represents some physical or abstract quantity or quantity.
	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.
	timestamp	some DDMS time	This is the most recent time at which the slot was updated.
subobject slot	value status	same	No slot of subobject was updated in this workspace.
		different	Some slot of subobject was updated in this workspace.
	timestamp	some DDMS time	This is the most recent time at which slots were updated.
set-valued slot	timestamp	some DDMS time	This is the most recent time at which a member of the set was created or destroyed or that any member was updated.

Table 1: Annotated Design Objects

datum	annotation	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 undestroyed in this workspace.
		destroyed in superior; undestroyed	The set member formerly existed in the superior workspace, was destroyed in that workspace, but has been undestroyed 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 annotations appropriate to the type of the member, as described by this table	described in this table	explained in this table
object reference slot	value	OID of object or $\perp$	The object ID of the object referenced, or a null value if there is no reference.
	value status	same	The reference in the slot was not changed in this workspace.
		different	The reference in the slot was changed in this workspace.
	timestamp	some DDMS time	This is the most recent time at which the slot was updated.

Table 1: Annotated Design Objects

datum	annotation	value	meaning
computed slot	validity status	void	Value of computed slot is not current and must be recomputed.
		valid	Value of computed slot is current.
	voided	true	Computed slot has been voided since workspace was last committed.
		false	Computed slot has not been voided since workspace was last committed.
	validated	true	Computed slot has been recomputed since workspace was last committed.
		false	Computed slot has not been recomputed since workspace was last committed.

	timestamp	some DDMS time	if (*validity status* = **void**)
			The earliest time that the slot was made void since it was last made valid.
			else
			The most recent time that slot was made valid.

	other annotations appropriate to the type of the computed slot, as described by this table	described in this table	explained in this table
derived slot	annotations appropriate to the type of the derived slot, as described by this table	described in this table	explained in this table

Programmatic Interface

CommitWorkspace(ToolID, WorkspaceID)

CommitWorkspace uses object annotations to compute the update delta between a workspace and its superior, then atomically applies the update delta to the superior. The root workspace W_{root}, since it has no superior, cannot be committed. A workspace $W \neq W_{root}$ can be committed only if the following two conditions are met:

1. The constraint requirements of Superior(W) are met by all objects in W.
2. There are no unresolved conflicts in W. The existence of an unresolved conflict indicates a problem that has not been resolved, such as a possibly erroneous update to a design. Preventing W from committing in this situation will block the propagation of errors to more public workspaces.

AbortWorkspace(ToolID, WorkspaceID)

As mentioned above, instead of committing a number of updates in a workspace W to its superior workspace Superior(W), the updates can be undone by aborting the workspace. A workspace can be aborted only if the following two conditions are met:

1. there are no tools which have W or a sub-workspace selected
2. there are no uncommitted updates in any sub-workspace of W

The abort operation is accomplished by discarding all annotated objects in W. Aborting a workspace W has no effect on Superior(W). After the abort, W and Superior(W) offer the same view of design objects.

Aborting a workspace is a rather drastic operation. A less drastic way to undo selected updates to a workspace is to use a tool to selectively perform *compensating updates* to achieve a desired state of design objects.

Section 4.B.6: Design Object Check-out and Check-in

A workspace can be thought of as the working area for a long transaction whose lifetime spans tool invocations. The DCM permits more than one tool, possibly under the control of multiple agents, to share updates to the same design object in the same workspace.

Before a tool can access a design object, it must check-out that object. The object can be checked-out for either read or update access. The DCM dramatically increases the potential for concurrency and cooperation in the design environment; it offers mechanisms which enable check-out of design objects without use of exclusive locks. Neither check-out for read nor check-out for update excludes check-out by other tools.

When a tool checks-out a design object X, the DCM returns the current state of the annotated object; the tool places that annotated object in its cache of objects. The DCM will send notifications of any updates made to X to the tool until the tool checks-in X. When a tool requests to check-out a design object X for update, it implicitly checks-out all Dep(X) for update access. (Src and Dep are defined in Section 2.D.2.) The tool must check-out Dep(X) because updates to X may affect computed slots in Dep(X).

A design object can be checked-out for read without any restrictions. In order for the DDMS to maintain a consistent view of objects, there are some restrictions on circumstances in which a tool is permitted to check-out a design object for update.

Restrictions on Checking-out Design Object for Update

A tool T which has selected workspace W can check-out object X (and implicitly Dep(X)) for update only if all of the following four conditions are met:

1. A supervisor has given the engineer who is running T the right to update Dep(X). If an engineer has been given permission to update X, but not to update some design object in Dep(X), then the engineer must create a new version of X; in this case an engineer

who does have permission to update $Y \in Dep(X)$ must integrate the new version of X into Y at some later time.

2. Each design object $Y \in Dep(X)$ is the latest version of a design element. Only the latest version of a design element can be updated. Obsolete versions are read-only. The DCM must be made aware of the creation and destruction of versions of design elements, so that it can ensure that this restriction is enforced. The way this is done is presented in Section 4.B.7.

3. For every design object $Y \in Src(X) \cup Dep(X)$, Y is not checked-out for update except in workspace W. This guarantees the invariant that if $X \Rightarrow Y$ and both X and Y are checked-out for update, then they are checked-out in the same workspace. See Figure 26.

4. For every design object $Y \in Src(X) \cup Dep(X)$, there are no uncommitted updates to Y in any workspace W' unless $W \leq W'$. This guarantees the invariant that if $X \Rightarrow Y$, 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. See Figure 27.

Since $X \in Src(X)$ and $X \in Dep(X)$, restriction 3 implies that an object can be checked-out for update in at most one workspace. If there is a need for two tools to update X at the same time, they must check-out X in the same workspace. Restriction 4 implies that a design object can be checked-out only in the same workspace or in a sub-workspace wherein there are uncommitted updates to the design object.

Suppose a tool submits updates $\Delta = < u_1, u_2, ..., u_n >$ to workspace Z at time t_{update}. Just prior to t_{update}, at time $\bar{t}$, for every workspace $W \neq W_{root}$, there exists some update delta $\Delta_W(\bar{t})$ such that:

$$V_W(\bar{t}) = V_{Superior(W)}(\bar{t}) + \Delta_W(\bar{t}).$$

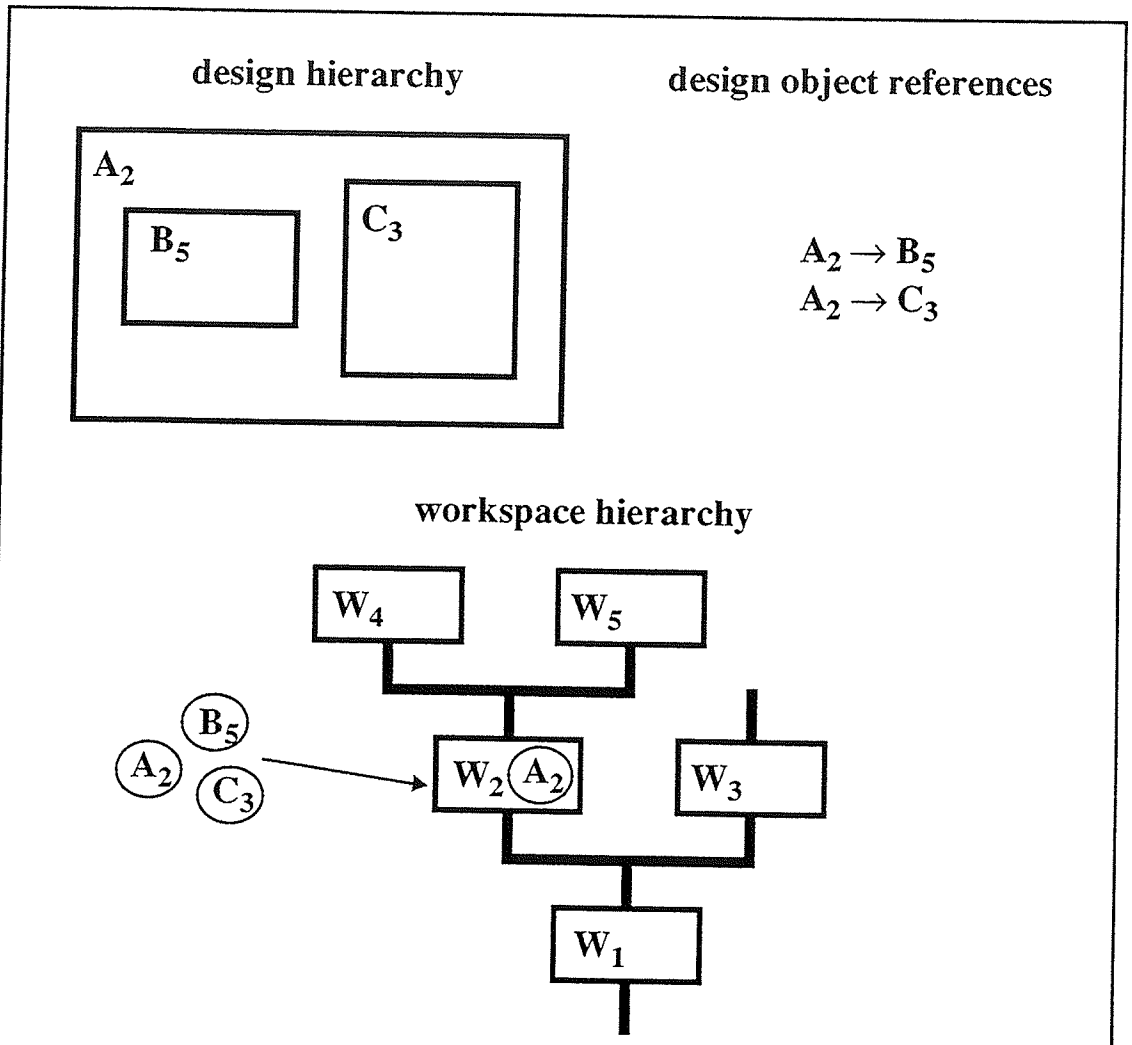

Figure 26: The third restriction on design object check-out.
If some tool has design object A_2 checked-out for update in workspace W_2, then other tools can check-out subdesigns A_2, B_5, and C_3 for update only in workspace W_2.

Invariants maintained by the last two restrictions guarantee that for each workspace $W \leq Z$:

$$V_W(t_{update}) = V_{Superior(W)}(t_{update}) + \Delta_W(t_{update}),$$

where $\Delta_W(t_{update}) = \Delta_W(\bar{t}) + \Delta$.

This result holds whether the update Δ comes from a committing inferior workspace or from a tool.

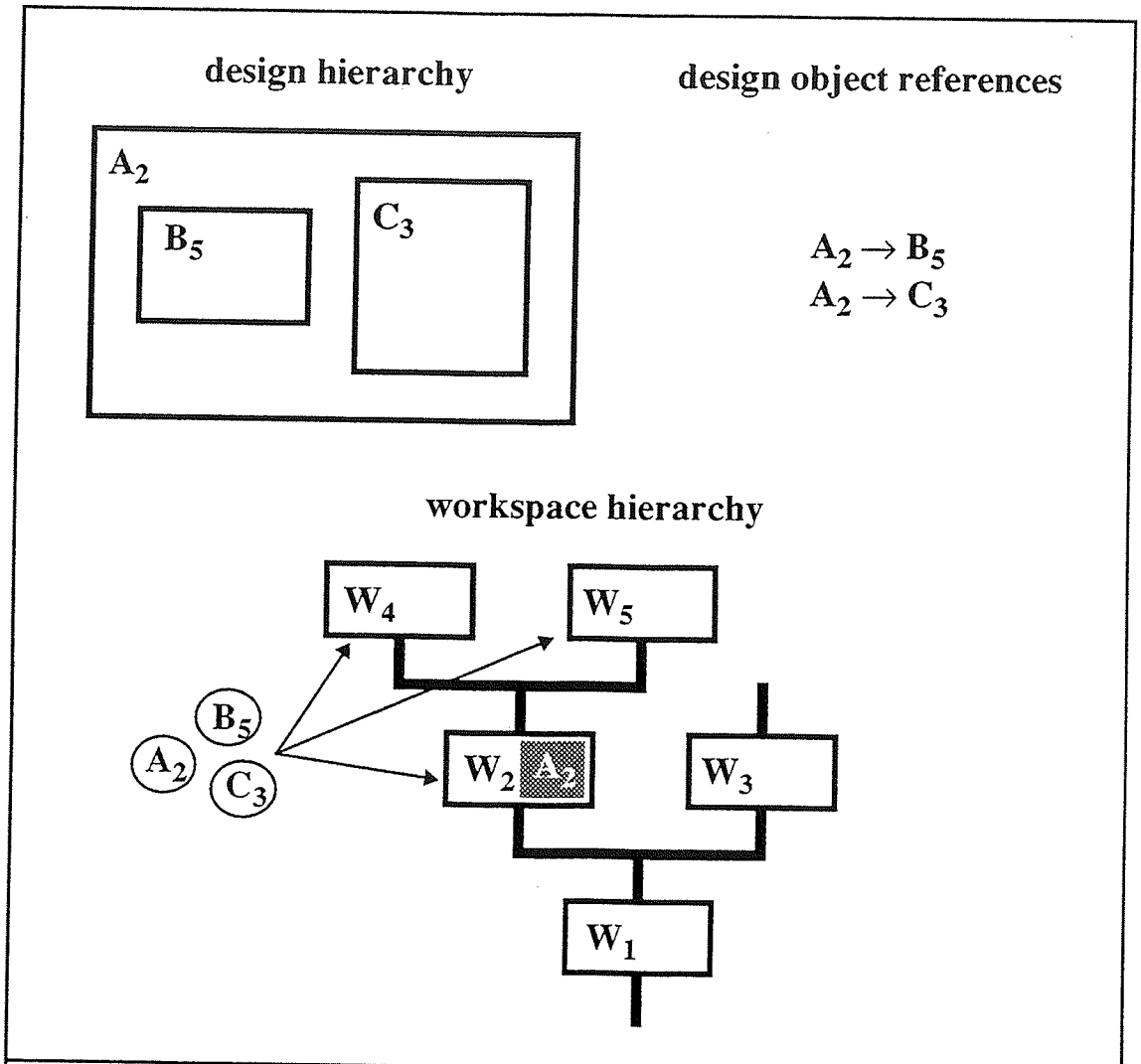

Figure 27: The fourth restriction on design object check-out.
If there are uncommitted updates to design object A_2 in workspace W_2, then tools can check-out design objects A_2, B_5, and C_3 for update only in workspaces W_2, W_4, or W_5.

Without these two restrictions the DCM 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 DCM is unable to merge updates, because this would require that it understand the semantics of the design data and the intent of the tool in making the update.

When a tool updates a design object to reference another, it does so within its object cache, then sometime later commits that reference to the workspace selected by the tool. The

DCM must be aware of a tool's intention to update a design object to reference another, so that it can ensure that these restrictions are enforced should the tool commit its updates. The way this is done is presented in Section 4.B.8.

Update Notifications

A tool checks-out into some workspace, and caches within its object cache, some number of design objects that it needs to work with. The DCM sends the tool update notifications of changes to all objects checked-out so that it can keep its cache consistent with the DDMS.

An update notification contains the following four pieces of information:

1. the ToolID of the tool that submitted the update and caused the notification to be sent
2. the DesignObjectID of the design object updated
3. the update operation which was applied to the design object (one of those described in Part 2.F)
4. a timestamp that records the DDMS time when the update was performed

A tool may defer handling those notifications so that the view of the design presented to the engineer does not change unexpectedly. In this case, the notifications will be handled later; in the meantime the view presented will be consistent, although somewhat out-of-date.

Over time, a tool may check-in some of the design objects it has checked-out or may check-out additional design objects. Consider a tool that checks-out and caches 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 tool. Then the state of objects in the tool's cache has become inconsistent, since updates to some design objects previously cached have not been incorporated into the cache; the object just checked-out, however, does already have the most recent updates applied to it. In order to prevent this

inconsistency from occurring, when a tool requests to check-out a design object, it passes the timestamp of the last update notification handled. If the design object to be checked-out has a timestamp later than that value, the request by the tool is refused. In this case it must handle additional notifications and resubmit the request if it chooses to do so.

Programmatic Interface

CheckOutForRead(ToolID, DesignObjectID, LastNotificationHandled)
returns annotatedObject / handleNotifications
CheckOutForUpdate(ToolID, DesignObjectID, LastNotificationHandled)
returns annotatedObject(s) / handleNotifications

A tool calls CheckOutForRead or CheckOutForUpdate to check a design object for read or update access, respectively. The Change Manager returns a copy of the annotated object or, in the case of CheckOutForUpdate, checks-out and returns all dependents of that object. Until it is checked-in, asynchronous notifications of updates to the design object will be sent from the Change Manager to the tool.

CheckIn(ToolID, DesignObjectID, LastNotificationHandled)
returns ok / handleNotifications

A tool calls CheckIn to inform the DCM that it no longer needs to access a design object. The Change Manager will then send no more notifications of updates to that object to the tool.

Section 4.B.7: Create or Destroy Version of Design Element

As explained in Section 4.B.6, the DCM must be kept aware of the creation of new versions of a design element so that it can mark other versions as obsolete and read-only. This is accomplished by having each tool make a request to the DCM when it needs either to create a new version or to destroy an existing version of a design element.

Update of Existing Version v. Creation of New Version

A tool can choose between updating an existing version of a design element or creating a new version. Suppose design object A_2 uses an instance of design object B_5 as a component and that B is to be updated. See Figure 28. A new version of B should be created if it is

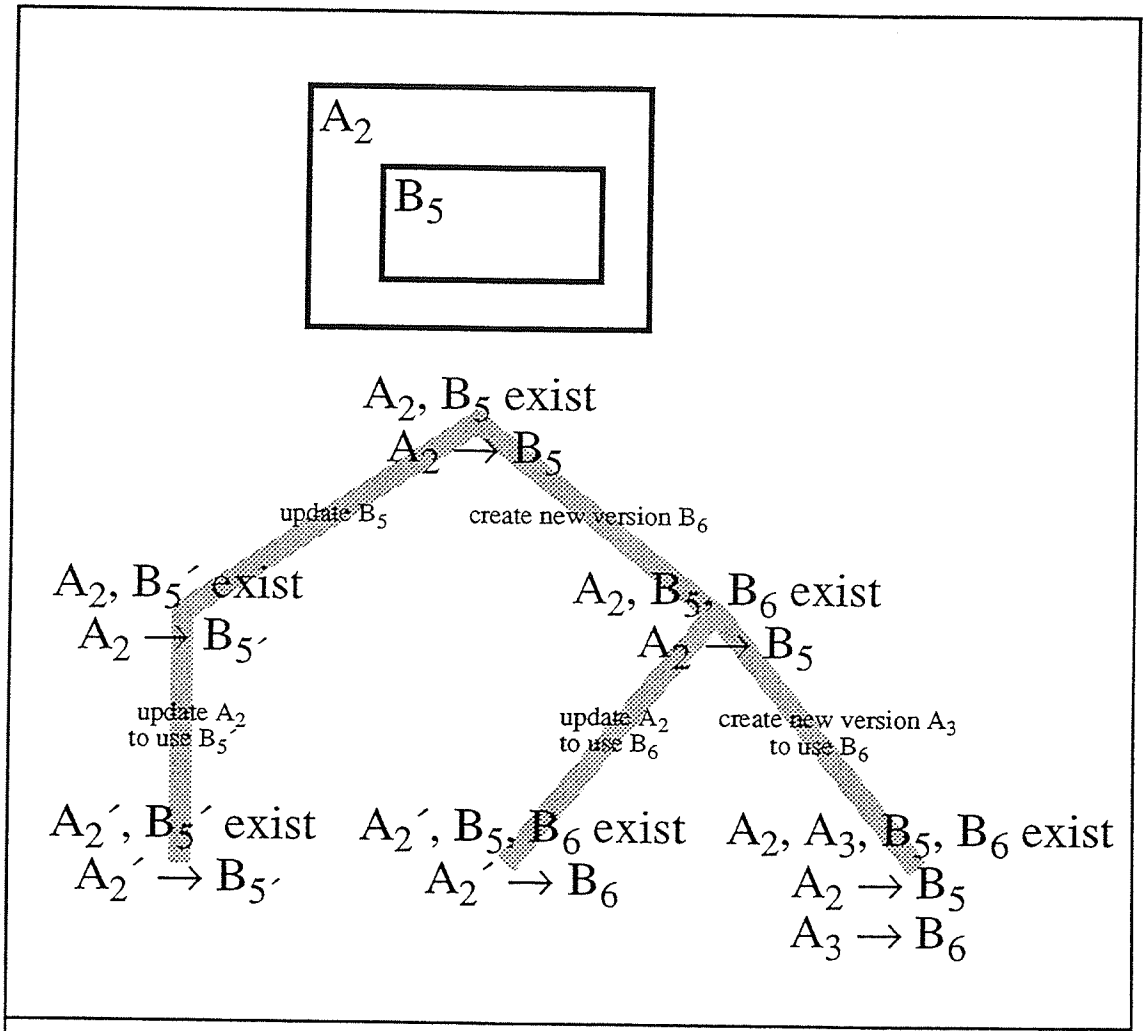

Figure 28: Update of existing version versus creation of new version of design object. This figure shows the tree of choices available to an engineer when updating a subdesign of a parent design.

desired not to affect A_2 and other design objects which depend upon B_5; creating a new version preserves existing versions and configurations of existing versions.

The disadvantage of creating a new version of B is that integration of B_6 into A_2 and other design objects using B has been deferred. Updates to B_6 which cause unanticipated difficulties in integration will not be identified until a later time. By updating an existing version an engineer see what impact his changes have on parent designs and what constraints are invalidated in those parent designs. Updating B_5 forces changes to B to be integrated into $Dep(B_5)$; an engineer is free, however, to make tentative updates to B_5 in an inferior workspace to see what effect those updates will have on design objects referencing B_5.

There are other approaches which automate particular policies of version control by using heuristics and rulebases [Björnerstedt 88]. The Change Manager can be used as the mechanism with which to implement such policies.

Programmatic Interface

CreateDesignElement(ToolID, ObjectType)
returns DesignObjectID

A tool calls CreateDesignElement in order to create version #1 of a new design element. The tool specifies what type of object is desired; that type must be one of the object types declared in the schema. CreateDesignElement creates a design object with default values for the slots and returns the OID of that object to the tool.

CreateVersion(ToolID, DesignObjectID)
returns DesignObjectID / notAllowed

A tool calls CreateVersion to inform the DCM that it wants to create a new version of a design element in the workspace W it has selected. The request will fail if the current latest version is checked-out for update by any tool, or if there are uncommitted changes to any version in any workspace other than W or super-workspaces.

DestroyVersion(ToolID, DesignObjectID)
returns ok / notAllowed

A tool can destroy an existing version of a design element by calling DestroyVersion. A design object X can be destroyed only if the following three conditions are met:

1. A supervisor has given the engineer using the tool permission to destroy the design object.
2. No other design object references X, in order to guarantee referential integrity.
3. X is not checked-out by any tool. Without this restriction an engineer might find that a design object being accessed is unexpectedly destroyed.

A design element ceases to exist when all of its versions have been destroyed.

Section 4.B.8: Adding and Removing Object References

As explained in Section 4.B.6, the DCM must be kept aware of references from one design object to another in a tool's cache that has not yet been committed to a workspace, so that the DCM can enforce the third restriction of checking-out a design object for update. This is accomplished by having each tool inform the DCM when it updates a design object so that it references, or no longer references, another design object.

Programmatic Interface

AddReference(ToolID, ObjectID, ReferencedObjectID)
returns ok / notAllowed

A tool informs the DCM that it has updated a design object in its cache to reference another design object by calling AddReference and identifying the referencing and referenced object. This call increments the reference count from the referencing to the referenced object. Suppose the tool has selected workspace W. This request will fail if the referenced object

either is checked-out for update in some workspace other than W or has uncommitted changes in any workspace W′ except where $W \leq W'$.

RemoveReference(ToolID, ObjectID, ReferencedObjectID)

A tool informs the DCM that it has updated a design object in its cache to no longer reference another design object by calling RemoveReference and identifying the referencing and referenced object. If the referenced object is no longer referenced by any object in the tool's cache, then it is free to be checked-out for update in workspaces other than the workspace selected by the tool, subject to the restrictions on checking-out a design object for update which were presented in Section 4.B.6.

Section 4.B.9: Update Design Objects in the DDMS

Each tool is free to submit a batch of updates to the workspace it has selected at any time. An update is any operation on an object as described in Part 2.F. The batch is applied atomically. A side effect of a tool's updates is that the DCM sends asynchronous update notifications to every other tool that has checked-out affected design objects in this or subworkspaces.

Handshaking to Eliminate Race Condition

Consider the following scenario. Assume there are two tools T_1 and T_2 working in the same workspace in the DDMS, sharing updates on object X with initial state S. Tools T_1 and T_2 initialize their internal state based on S. Tool T_1 applies update u_1 to its copy of X, yielding $S_1 = S + \Delta_1$, and T_2 applies update u_2 to its copy of X, yielding $S_2 = S + \Delta_2$. Suppose that T_1 and T_2 now submit u_1 and u_2 to the DDMS, and the DCM sends relevant notifications to T_2.

Suppose that, due to network delays, update u_2 now arrives from tool T_2. The DDMS doesn't know whether T_2 received the notification before or after submitting u_2. If T_2 sub-

mitted u_2 before processing the notification, then the DDMS should reject u_2, since T_2 may have predicated its update u_2 on X having state S rather than state S_1 as it does now. On the other hand, if T_2 did process the notification, then update u_2 should be applied to X. Thus a race condition exists between the receipt of an update request from a tool and the delivery of an update notification from the DCM in the DDMS.

The DCM eliminates this race condition by using the following handshaking protocol: Each update notification is tagged with the current DDMS time, and the DCM remembers the timestamp of the most recent notification sent to each tool. Whenever a tool submits an update request, the timestamp of the last notification it processed is included. The DCM then compares that timestamp with that of the last update notification sent to the tool. If an update notification was sent since the time the update request was sent, then the DCM knows that the tool's update request was based upon incomplete knowledge. In this case the update request is rejected by the DDMS and an error code is returned to the tool which indicates that the tool has failed to process all update notifications. In response, the tool must handle the notifications sent and resubmit the update request if it so chooses. This situation may repeat—by the time the tool resubmits its request, there may be additional notifications that it must handle before the DCM will accept the request.

Programmatic Interface

UpdateWorkspace(ToolID, **list** Update, NotificationTimestamp)
returns time/ handleNotifications / invalidConstraint

A tool commits its updates to the workspace it has selected by calling UpdateWorkspace. If the requests succeeds, the DDMS returns the current time. The tool uses the current DDMS time to alter timestamps in its annotated object cache, as explained in Section 5.B.8. When a tool commits its changes to the DDMS, the DDMS decrements counts of uncommitted references between design objects; the purpose of these reference counts is explained in Section 4.B.8. UpdateWorkspace will fail either if the tool has not handled all the

update notifications sent by the DCM or if accepting the updates would cause one or more of the workspace's constraint requirements not to be satisfied.

Section 4.B.10: Conflict Logging

As explained in Section 3.B.9, conflict logging is needed in a design environment for concurrent design. The Conflict Logger in the DCM offers that service to tools.

Programmatic Interface

LogConflict(ToolID, OffendingToolID, Complaint)
returns ConflictID

A tool invokes LogConflict to register a complaint about an update made by another tool. The tool identifies the offending tool and supplies a textual explanation of how the update constitutes a conflict. The DCM doesn't understand the semantics of conflicts, and can make no attempt to remedy the conflict; it merely provides the *mechanism* with which conflicts can be recorded. Higher-level mechanisms are responsible for implementing particular *policy* [Hall 89].

Conflicts are logged in the workspace which the complaining tool has selected. If a workspace has unresolved conflicts, it is not allowed to commit, as explained in Section 4.B.5.

ResolveConflict(ToolID, ConflictID, Resolution)

After some action has been taken to remedy a conflict, it can be marked as having been resolved; a textual explanation of how the conflict was resolved is supplied. Engineers who cannot resolve a conflict may need assistance from their supervisor in order to do so. After all conflicts are resolved, updates in a workspace can be committed to the superior workspace. The DCM guarantees that registered conflicts are not lost, but provides no assurance that a responder handles the resolution of a conflict correctly.

Section 4.B.11: Active Design Status

As discussed in Section 3.B.10, knowledge of the design status can aid engineers in the task of planning and coordinating their activities. The DCM provides the mechanism with which tools and the agents using them can access and stay aware of changes to the design status. Higher level mechanisms can use this mechanism in order to implement policies of design methodology or shared access.

The design status made available to tools by the DCM is exactly that internal state of the DCM that can be altered by invocations of the procedures whose interfaces were presented in this chapter, excluding updates to design data:

- which tools are currently running
- what is the hierarchy of workspaces
- which tools have selected what workspaces
- the constraint requirements attached to a workspace
- what workspaces have uncommitted updates
- which design objects have been checked-out by what tools
- what versions a design element has
- which design objects reference which other design objects
- what conflicts and conflict resolutions have been logged in a workspace

Programmatic Interface

```
GetDesignStatus( ToolID, WhichDesignStatus )
    returns DesignStatusRequested
```

Tools can obtain any of the above design status by calling GetDesignStatus and specifying what design status is needed.

RegisterInterestInDesignStatus(ToolID, WhichDesignStatus, WhichChange)
returns InterestID

At times it is useful for a tool not only to obtain design status, but to track changes to it. For example, an engineer might like to know when another engineer has checked-out the same design object, or that a new version of some design object is available. Similarly, a supervisor may wish to monitor what tools are run in what workspaces.

The design status monitor in the DCM makes the design status **active**, that is, the design status can perform some action, such as sending a notification to a tool, when it is changed. A tool indicates of what changes in design status it would like to be notified by calling RegisterInterestInDesignStatus and specifying in what change to which design status it is interested. RegisterInterestInDesignStatus returns an ID of the interest registered.

Until an interest which has been registered is cancelled, the tool will receive asynchronous notifications of changes to the specified design status. Those notifications travel along a different channel of communication (a different socket for IPC, for example) from the one used by the Update Monitor when it notifies the tool of updates to design objects it has cached. This is done so that a tool can keep a current view of the design status while deferring the handling of update notifications.

UnregisterInterestInDesignStatus(ToolID, InterestID)

UnregisterInterestInDesignStatus cancels an interest that a tool had registered, and indicates to the Design Status monitor that the tool no longer wishes to receive notifications of those changes to the design status specified by the interest.

Part 4.C: Invariants Maintained by the DDMS Change Manager

All modules within the DDMS Change Manager work together to jointly provide a collection of services to CAD tools. Guaranteeing internal consistency and correct operation of the DCM requires that it maintain a number of invariants. This part of the chapter summarizes those invariants.

Section 4.C.1: Workspace Related to Superior by Update Delta

CAD tools submit batches of updates to workspaces in the DDMS. In addition, updates in workspaces may be committed to their superior workspace. The DCM guarantees:

Invariant #1:

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

Then $V_W(t) = V_{Superior(W)}(t) + \Delta(t)$, where Δ 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 design objects in W by CAD tools, and (2) updates to design objects in W resulting from inferior workspaces of W having committed to W.

In order to preserve this invariant, the DCM enforces four restrictions on when design objects may be checked-out for update by tools; these restrictions ensure that updates applied to a workspace have no effect on the states of objects in subworkspaces. The four restrictions are presented in Section 4.B.6.

Section 4.C.2: Annotations Provide Sufficient Information for Commit

A CAD tool can request that updates to a workspace be committed to its superior workspace at any time. When it does so, the DCM scans the annotated object cache, interprets

the annotations to determine how the objects were updated, and generates a list of updates which are then applied to the superior workspace.

The invariant which makes this possible is:

Invariant #2:

The update delta can be computed at any time from the annotations on objects.

This invariant is maintained by the DCM because the DCM alters the annotations on objects, as well as the values of object slots, whenever it processes an update request from a tool or from an inferior workspace. The annotations in a workspace after each update reflect how the states of design objects differ between that workspace and the superior workspace. Table 1 in Section 4.B.5 describes the annotations in detail. Table 2 in Section 5.B.4 explains how the annotations are altered as a result of updates being applied to design objects. Table 5 in Section 5.B.8 shows what updates result when the DCM scans object annotations in a workspace in order to compute the update delta to be applied to the superior workspace.

Section 4.C.3: Unresolved Conflicts Restrict Workspace Commit

If there are unresolved conflicts in a workspace W, the DCM prevents W from committing to its superior workspace Superior(W) so that potentially erroneous or deleterious updates are confined to W.

Invariant #3:

Workspaces with unresolved conflicts cannot be committed.

The DCM enforces this invariant by first checking for unresolved conflicts within a workspace before honoring a request to commit a workspace. Conflicts are discussed in Sections 3.B.9 and 4.B.10. Committing updates in a workspace is explained Section 4.B.5.

Section 4.C.4: Unhandled Notifications Restrict Updates By Tool

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

Invariant #4:

Update requests from tools will be honored only if the tool has handled all asynchronous update notifications which have been sent to it.

This invariant is preserved by virtue of the handshaking protocol used between the tool and the DCM which is based on sequence numbers for the notifications. This protocol is explained in Section 4.B.9.

It is important to note that this invariant does not guarantee that a tool has responded *in a proper fashion* to the notifications it has received. Such a guarantee of the tool's behavior cannot, in general, be enforced by the DCM. There is some assistance given to applications by the Change Manager in the tool, however, in keeping the object cache consistent with the DDMS; this is explained in Section 5.B.5.

Section 4.C.5: Constraint Requirements are Met

As explained in Sections 3.B.8 and 4.B.4, constraint requirements specify a degree of consistency which is to be maintained at all times within a workspace. The DCM guarantees:

Invariant #5:

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

The Change Manager preserves this invariant by refusing updates to a workspace W, either from tools which have selected the workspace or from a committing inferior workspace of

W, in which one or more constraint requirements attached to W is not true (i.e., **false** or **void**). Sections 4.B.5 and 4.B.9 discuss committing updates in a workspace and in a tool's object cache, respectively.

Section 4.C.6: Only Latest Version of Design Object Can Be Updated

As explained in Section 2.D.3, all versions of a design object except the latest version are read-only and cannot be updated.

Invariant #6:
Only the latest version of a design object can be updated.

The Change Manager in the DDMS maintains this invariant by not allowing a design object which is not the latest version, nor any design object X with some $Y \in Dep(X)$ which is not the latest version, to be checked-out for update by any tool. Design object check-out is explained in Section 4.B.6. The DCM stays aware of what versions exist because a tool must inform the DCM when a version is created or destroyed; the way this is done is presented in Section 4.B.7.

Section 4.C.7: Referential Integrity is Enforced

The DCM enforces referential integrity within the object store:

Invariant #7:
A design object cannot be destroyed if there are any references to it from other objects.

The DCM maintains this invariant by first checking whether an object X is referenced by any other object before honoring a request from a tool to destroy X. This includes checking whether X is referenced by another object within the object cache of any tool. The DCM knows which objects reference other objects within caches of tools, because a tool must notify the DCM when it adds or removes an object reference within its cache; the way this is done is presented in Section 4.B.8.

Chapter 5

Change Manager Support for CAD Tools

The CAD tool of the future consists of application code, along with application-specific data structures and algorithms, plus additional functionality provided by the Tool Change Manager (TCM). This chapter presents the architecture of a CAD tool, describes the functionality added by the TCM, presents the programmatic interface between an application and the TCM, and summarizes the invariants maintained by the TCM.

Part 5.A: Architecture of a CAD Tool

Section 5.A.1: Introduction

A CAD tool consists of an application plus the Tool Change Manager. The application implements the particular functionality of the tool, and is what distinguishes one tool from another. A tool developer is responsible for developing an application and linking it with the TCM. The TCM consists of six modules: tool clock, object cache manager, void propagator, update notification handler, interest matcher, and dependency locator. See Figure 29. These six modules work together and offer a collection of services to an application. Note that an application does not communicate directly with the object store; it performs updates only through the TCM, which then communicates with the DCM.

The TCM is linked with the application to create a CAD tool; because of this, communication between the application and the TCM is inexpensive. Thus frequent interaction be-

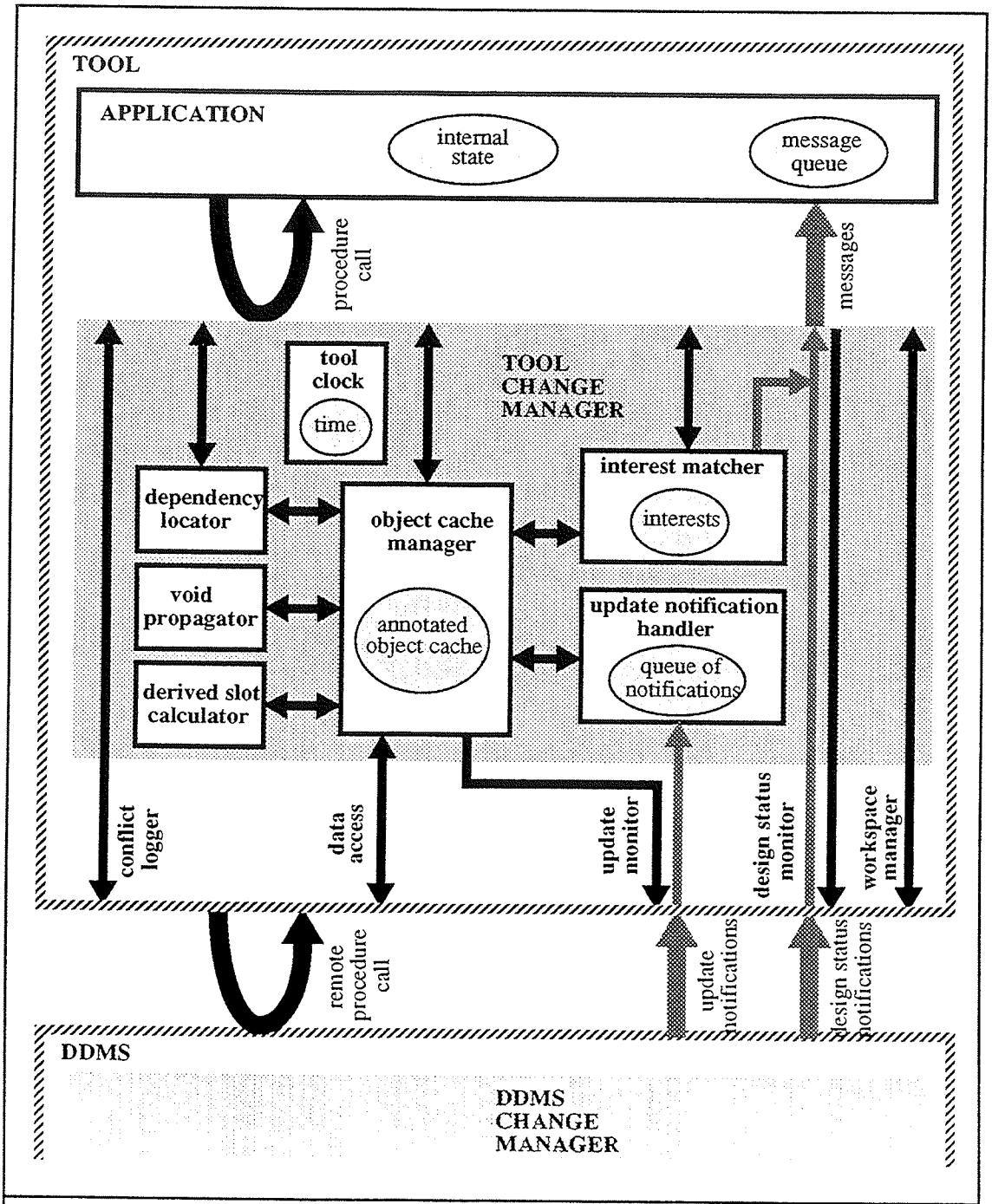

Figure 29: Architecture of a CAD tool.

A CAD tool consists of the application, which is responsible for implementing the functionality of the tool, such as design synthesis, editing, or consistency checking, plus the Tool Change Manager. The Tool Change Manager is a layer between CAD tools and the application; it consists of seven modules: tool clock, object cache manager, void propagator, derived slot calculator, update notification handler, interest matcher, and dependency locator.

tween the application and the TCM is not inefficient, and the granule of interaction can be small—there are operations involving slots of design objects, as compared to entire objects, which is the case between CAD tools and the DDMS.

Section 5.A.2: Tool Clock

The TCM maintains an integer-valued clock in each tool. The clock in each tool is separate from, and run asynchronously with respect to, the clock in the DCM and the clocks in other tools. The tool clock represents the amount of time elapsed since a tool last committed its updates to the workspace it has selected. The tool clock is initialized to zero when a tool selects a workspace. It is incremented whenever the TCM processes any update request from the application. The tool clock is reset whenever the tool commits its changes. The times at which slots of design objects change value are stored in the annotated object cache, which holds the same information as does the annotated object store in the DDMS. Section 4.B.5 explains what annotations are attached to objects.

Section 5.A.3: Message Queue

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

Section 5.A.4: Object Cache Manager

Within each tool is a cache of annotated design objects which the tool has checked-out and is currently accessing and manipulating. The cache is similar to a workspace in the DDMS in that all updates are encapsulated within the cache; the updates are applied atomically to the workspace the tool has selected when the tool commits its changes. Use of a cache en-

ables a tool to make experimental updates to local copies of objects. Unlike workspaces in the DDMS, the lifetime of the annotated object cache is tied to that of the tool. The **object cache manager** within the TCM gives an application access to the annotated object cache by handling requests both to load data not yet cached and to update the cache.

Part 5.B: Functionality of the Tool Change Manager

This part of the chapter describes each service that the Tool Change Manager offers to applications, presents the programmatic interface which an application uses to access the service, and explains how modules within the TCM operate in order to provide that service.

Section 5.B.1: Services from DDMS

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

AppStart(AgentName, ToolName)

When an application begins operation it must notify the TCM. It does so by calling AppStart. In response, the TCM initializes itself and registers the tool with the DCM by calling RegisterTool. The TCM remembers the ToolID returned by RegisterTool for use in subsequent requests to the DDMS.

AppShutDown( )
    returns ok / workspaceSelected

An application must also notify the TCM when it wishes to end operation. It does so by calling AppShutDown. The TCM permits shutdown only if the application has no workspace currently selected. The TCM calls UnregisterTool in the DCM when the application shuts down.

- creating and destroying workspaces
- workspace selection
- constraint requirements
- committing and aborting workspaces
- creating and destroying versions of design elements
- conflict logging
- active design status

The TCM contains procedures that provide the above services to an application. These procedures provide the same functionality that the DCM offers to tools, as detailed in Part 4.B. Invoking any of them calls the DCM procedure of the same names. The programmatic interfaces of these procedures differ only in that the application need not pass a ToolID when it calls procedures of the TCM; the ID of the tool is passed automatically by the TCM to the DCM on behalf of the application.

Section 5.B.2: Design Object Check-out and Check-in

An application requests that a design 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 DDMS will send asynchronous notifications of updates made by other tools to that object to the **update notification handler** in order that the cache be made consistent; the way this is done is explained in Section 5.B.5. A design object will remain cached and accessible to the application until the application checks-in the design object.

Programmatic Interface

AppCheckOutForRead(ToolID, DesignObjectID, LastMessageHandled)
returns annotatedObject / handleMessages
CheckOutForUpdate(ToolID, DesignObjectID, LastMessageHandled)
returns annotatedObject(s) / handleMessages

An application calls AppCheckOutForRead or AppCheckOutForUpdate to check a design object for read or update access, respectively. The TCM calls CheckOutForRead or CheckOutForUpdate in the DDMS, respectively, and returns to the application a copy of the annotated object or, in the case of CheckOutForUpdate, checks-out and returns all dependents of that object.

The same restrictions on a tool in checking-out an object for update, as presented in Section 4.B.6, apply to an application. There is one difference between AppCheckOutForRead (or AppCheckOutForUpdate) offered by the TCM to an application and CheckOutForRead (or CheckOutForUpdate, respectively) offered by the DCM to tools: an application passes the timestamp of the last message it handled to AppCheckOutForRead; the TCM converts this timestamp to the timestamp of the last notification handled by the tool and passes that to CheckOutForRead.

AppCheckIn(ToolID, DesignObjectID, LastMessageHandled)
returns ok / uncommittedUpdates / handleMessages

An application calls AppCheckIn to inform the TCM that it no longer needs to access a design object. The TCM calls CheckIn in the DCM so that it will then send no more notifications to the tool of updates to that object. AppCheckIn will fail if the application has failed to handle all messages sent to it or if, in the case of a design object checked-out for update, there are uncommitted updates to the design object in the cache.

Section 5.B.3: Read Design Objects in Cache

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

Programmatic Interface

GetSlotValue(OID, SlotName)
returns value

The value returned by GetSlotValue depends upon the type of the slot:

primitive slot: value of the slot (of the appropriate type, as determined by the schema)

subobject: OID of the subobject

set-valued slot: values (or OIDs in the case of subobjects) of members of the set

object reference slot: OID of the referenced object

computed slot: the value of the computed slot if it is valid, else ⊥ (undefined) if the computed slot is void

There is a special procedure GetStaleSlotValue that returns the old value of a computed slot. The old value can be used by an application that is incrementally recomputing the new value of the computed slot; incremental computation is explained in Section 5.B.6.

derived slot: the value of the slot (of the appropriate type, as determined by the schema)

Section 5.B.4: Update Design Objects in Cache

An application effects change in design data by updating design objects in the cache, then committing those updates to the workspace it has selected. This section discusses how computed slots are marked **void**, presents the programmatic interface with which an application updates cached design objects, and describes what effect each update operation has on the annotated object cache and on other parts of the TCM.

Voiding of Computed Slots

When slots in a design object X are updated, computed slots in X and in other objects that reference X may be invalidated. It is unreasonable to assume that every application consci-

entiously voids computed slots whenever it updates slots in the object cache which may affect the computed slots. Furthermore, updates to the annotated object cache by the update notification handler (presented in Section 5.B.5) may affect the validity of computed slots.

For these reasons, a truth maintenance system [Doyle 79] in the TCM called the **void propagator** performs the task of voiding computed slots in the annotated object cache whenever slots upon which they depend, as defined by the schema, are updated by either the application or the update notification handler. The act of voiding a slot removes any value that the computed slot had and assigns a default value to it. Note that voiding a computed slot is different from computing the new value of the computed slot. The TCM doesn't know how to recompute the value of a computed slot; instead, that is the responsibility of applications, and may require an arbitrary amount of computation. Computed slots are explained in Part 2.E.

One update may have a ripple effect in which computed slots, and other computed slots affected by those computed slots, are affected. The void propagator recursively marks computed slots affected by the update as **void**. Suppose an update to design object X caused the void propagator to be invoked. The void propagator is guaranteed to have update access to all design objects affected, since when X was checked-out for update so were Dep(X); this side effect of design object check-out was explained in Section 5.B.2.

The computational complexity of void propagation depends upon the number of computed slots which depend upon an updated slot. In design data this would correspond to the number of times a design object was used as a subdesign in parent designs, the number of times those parent designs were used as subdesigns in other designs, etc. Thus modifying a low-level design such as flip/flop in a layout may cause a large number of computed slots to be marked as **void**. It is rare, of course, that such a drastic change would be made late in the design process; an engineer making such a change would expect that change to have potentially far-reaching consequences.

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

Recomputation of Derived Slots

Just as the value of a computed slot can become stale when a slot upon which it depends has been updated, so can the value of a derived slot. Instead of calling the void propagator merely to mark the derived slot **void** as it does a computed slot, the TCM invokes the **derived slot calculator** to recompute the value of the derived slot. Derived slots are explained in Part 2.E. The derived slot calculator computes the value of the derived slot based upon the specification in the schema. Part 2.C discusses the object schema.

Programmatic Interface

An application creates and destroys design objects, which are versions of design elements, by calling CreateDesignElement, CreateVersion, and DestroyVersion in the DDMS; these functions are explained in Sections 4.B.7 and 5.B.1. An application updates a design object by calling procedures in the object cache manager. When an application updates a design object the tool clock is incremented; the time that an update occurs is used to timestamp updated slots.

Table 2 below shows the update procedures that can be called for each type of slot in a design object and what effect each procedure has on the annotated object cache. Annotations are used when a tool commits its updates to the DDMS in order to determine the differences (that is, the update delta) between the object cache in the tool and the workspace which the tool has selected; Section 5.B.8 discusses how the object delta is computed.

Table 2: Procedures to Update Design Objects

type of slot	procedure	changes to annotated object cache, where t_{update} = TCM time of update from application
primitive	ChangeValue	*value* ← new value *value status* ← different *timestamp* ← t_{update}
set-valued	CreateMember	*existence status* ← not in workspace; created *timestamp* ← t_{update}
	DestroyMember	if (*existence status* = created in workspace; unchanged) then *existence status* ← created in workspace; destroyed else if (*existence status* = destroyed in workspace; undestroyed) then *existence status* ← destroyed in workspace; unchanged else if (*existence status* = not in workspace; created) then *existence status* ← not in workspace; destroyed *timestamp* ← t_{update} DestroyMember calls RemoveReference (presented in Section 4.B.8) for each uncommitted inter-object reference that is removed as a result of destroying the set member.
	UndestroyMember	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; undestroyed else if (*existence status* = not in workspace; destroyed) then *existence status* ← not in workspace; created *timestamp* ← t_{update} UndestroyMember calls AddReference (presented in Section 4.B.8) for each uncommitted reference that is added as a result of undestroying the set member.

Table 2: Procedures to Update Design Objects

type of slot	procedure	changes to annotated object cache, where t_{update} = TCM time of update from application
object reference	ChangeReference	*value* ← OID of object to reference or *value* ← ⊥ (nullify a reference) *value status* ← different *timestamp* ← t_{update} When an application adds a reference from one design object to another, the TCM calls AddReference in the DCM. When an application removes a reference the TCM calls RemoveReference in the DCM.
computed	MarkVoid	if (*validity status* = **valid**) then *validity status* ← **void** *timestamp* ← t_{update} *voided* ← **true** MarkVoid is called recursively by the void propagator.
	MarkValid	*validity status* ← **valid** *validated* ← **true** *timestamp* ← t_{update} MarkValid is called by an application after it recomputes and updates the value of a computed slot.
	procedures to update value of computed slot	Which procedures in this table can be invoked to update the value of a computed slot depends upon the type of the value of the computed slot, as specified by the schema. These procedures are called by an application that has re-computed the value of the slot.
derived	procedures to update value of derived slot	Which procedures in this table can be invoked to update the value of a derived slot depends upon the type of the value of the derived slot, as specified by the schema. These procedures are called only by the derived slot calculator and not by an application. To an application, the value of a derived slot always appears current.

The table above doesn't show the four side effects of every update:

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

   value status ← different

   timestamp ← t_{update}

2. The update will cause computed slots affected by the update to be voided by the void propagator.

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

4. The **interest matcher** will deliver a message to the application if the update matches an interest placed by the application. Interests are explained in Section 5.B.6.

The TCM enforces two restrictions on when an application can update a design object in the annotated object cache:

1. The application must have checked-out the object for update. Object check-out is explained in Section 5.B.2.

2. The application must have handled all messages sent to it by the interest matcher. Handshaking similar to that explained in Section 4.B.9 guarantees that this is done: When the application submits an update request to the object cache manager, it passes the timestamp of the last message it handled. If that timestamp is earlier than the timestamp of the last message sent from the interest matcher to the application, then the object cache manager knows that the application has failed to handle all messages and will refuse the update request.

Section 5.B.5: Handle Update Notifications

When a tool commits updates to design object X to workspace W in the DDMS (explained in Section 4.B.9), the cache within every other tool that has selected either W or a subworkspace of W and which has checked-out X will become stale. The update monitor in

the DCM uses triggers in the object store to guarantee that each tool that is caching X will be sent asynchronous notifications of all updates to X. Notifications are explained in Section 4.B.6.

When notification of an update by another tool is received from the update monitor, that update must be incorporated or "merged" into the tool's annotated object cache. This action is performed by the **update notification handler** in the TCM. Like the void propagator and the derived slot calculator, the update notification handler operates automatically on behalf of an application.

Purpose of the Update Notification Handler

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

Invariant: $V_T(t) = V_W(t) + \Delta(t)$, where $\Delta(t)$ is the update delta, or difference, from W to T. 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 handler, and that the update notification handler merges u into the object cache at time t_{merge}. It does so by altering V_T 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 handler restores the invariant by finding some small δ such that:

$V_T(t_{merge}) = V_W(t_{merge}) + \Delta(t_{merge})$,

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

$\Delta(t_{merge}) = \Delta(t_{merge}\text{-}1) + \delta$

Operation of the Update Notification Handler

When the update notification handler receives an update notification from the update monitor, it tries the best it can to effect the change in the object cache. Table 3 below shows how the update notification handler updates the annotated object cache for each type of update notification it can receive. As is the case with updates from an application, updates from the update notification handler can have side effects, as explained in Section 5.B.4.

Table 3: Operation of the Update Notification Handler

type of slot	**update notification**	**changes to annotated object cache, where t_{update} = DCM time of update in DDMS**
primitive	ChangeValue to v	*value* ← v *value status* ← same *timestamp* ← 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 handler will first call UndestroyMember to restore the object, then perform the update.
set-valued	CreateMember	*existence status* ← created in workspace; unchanged *timestamp* ← t_{update}
	DestroyMember	*existence status* ← destroyed in workspace; unchanged *timestamp* ← t_{update} The update notification handler calls RemoveReference (presented in Section 4.B.8) for each uncommitted inter-object reference that is removed as a result of destroying a set member.
	UndestroyMember	*existence status* ← created in workspace; unchanged *timestamp* ← t_{update} The update notification handler calls AddReference (presented in Section 4.B.8) for each uncommitted reference that is added as a result of undestroying a set member.
object reference	ChangeReference to X (or to ⊥)	*value* ← OID of X (or ⊥, if reference was nullified) *value status* ← same *timestamp* ← t_{update} When the update notification handler removes an existing reference from one design object to another, it calls RemoveReference in the DDMS.

Table 3: Operation of the Update Notification Handler

type of slot	update notification	changes to annotated object cache, where t_{update} = DCM time of update in DDMS
computed	MarkVoid	When the update notification handler incorporates other tools' updates into the object cache, the void propagator will automatically void any computed slots affected; no action need be taken by the update notification handler when it receives notification of a computed slot having been marked void.

	MarkValid	if (*voided* = **false**) then
		validity status ← **valid**
		timestamp ← t_{update}
		else
		validity status ← **void**
		If a tool T_1 recomputes and marks a computed slot as valid, that validity can propagate to the object cache of another tool T_2 only if T_2 has never voided computed slot in its cache.

	updates to the value of a computed slot	After a tool recomputes a computed slot, it updates the slot to contain the new value. Thus the update notification handler may receive notifications of updates by other tools to a computed slot. It responds by applying those updates, as described by this table, to the computed slot in the object cache.
derived	procedures to update value of derived slot	When the update notification handler incorporates other tools' updates into the object cache, the derived slot calculator will automatically void any computed slots affected; no action need be taken by the update notification handler when it receives notification of a derived slot having been updated.

It is important to note that the manner in which the update notification handler merges updates from other tools into the object cache is *syntactic* rather than *semantic*. The update notification handler does not understand any meaning which may be assigned to the state of design objects. Thus when the update notification handler merges updates it may unknowingly undo updates to or adversely affect the state of the object cache within the tool. *In such a case the application is responsible for applying compensating updates to the ob-*

ject cache in order to restore it to a "consistent" or "useful" state before committing the state of the object cache to a workspace in the DDMS. An application which does so is called "well-behaved"; this concept is covered in Chapter 6.

Deferred Handling of Update Notifications

In normal operation, the update notification handler makes asynchronous changes to the object cache in response to update notifications, received from the update monitor in the DDMS, that describe updates made by other tools. Thus the view of data presented to an application is subject to change. At times it may be convenient for an application to present an unchanging view of design data to an engineer, and for the processing of update notifications by the update notification handler to be deferred. For example, an engineer might choose not to be bothered by updates made by other engineers until the end of each day. Note that the disadvantage of deferring the incorporation of updates made by other engineers is that the engineer will not be aware of potentially conflicting or erroneous updates until the merging of updates resumes [Winslett 89]. But at that time other updates may have been predicated on the erroneous updates, and correcting the resulting problems will be more difficult. In general, identifying conflicts early rather than late in the design process reduces the cost of design [Boehm 81].

The TCM offers an application the ability to cause the update notification handler to defer or to resume the merging of update notifications into the object cache. For the period of time that the merging is deferred, the view of design data that the application sees may be stale.

When the object cache is stale, an application operates based on a view of the world that is somewhat incorrect. For this reason, the application is restricted in what it can do while the update notification processor has been turned off; in particular, it is not allowed to update design objects in the DDMS. The handshaking used in the procedures CheckOutForRead, CheckOutForUpdate, CheckIn, AppCheckOutForRead, AppCheckOutForUpdate,

AppCheckIn, and AppCommit prevent an application from checking objects out or in or from committing its updates to the DDMS unless the update notification handler has handled all update notifications and the application has handled all messages that result. These procedures are described in Sections 4.B.6, 5.B.2, and 5.B.8.

Programmatic Interface

DeferUpdateHandling( )

An application calls DeferUpdateHandling when it wants to suspend operation of the UpdateNotification handler. The application is then assured that any changes to the object cache are results of its updates, not those of other tools.

ResumeUpdateHandling( )

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

Section 5.B.6: Register Interests in Updates

After a tool checks-out (and caches) a design object X, the update monitor in the DCM sends notifications of any updates to X to the update notification handler in the tool, which uses the notification to make the object cache current. The application in a tool 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 handler 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 tool, 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 upon the particular application. The TCM does

not understand the semantics of application. Thus it is the application's responsibility to inform the TCM of which updates it needs to be informed. It does so by registering **interests** with the **interest matcher** in the TCM. Each interest identifies some set of updates. When any update specified by an interest occurs, the interest matcher sends a message to the application that describes the update. An application registers some number of interests with the interest matcher; the interests are chosen so that the set of updates in which the application is interested is covered by the interests. See Figure 30.

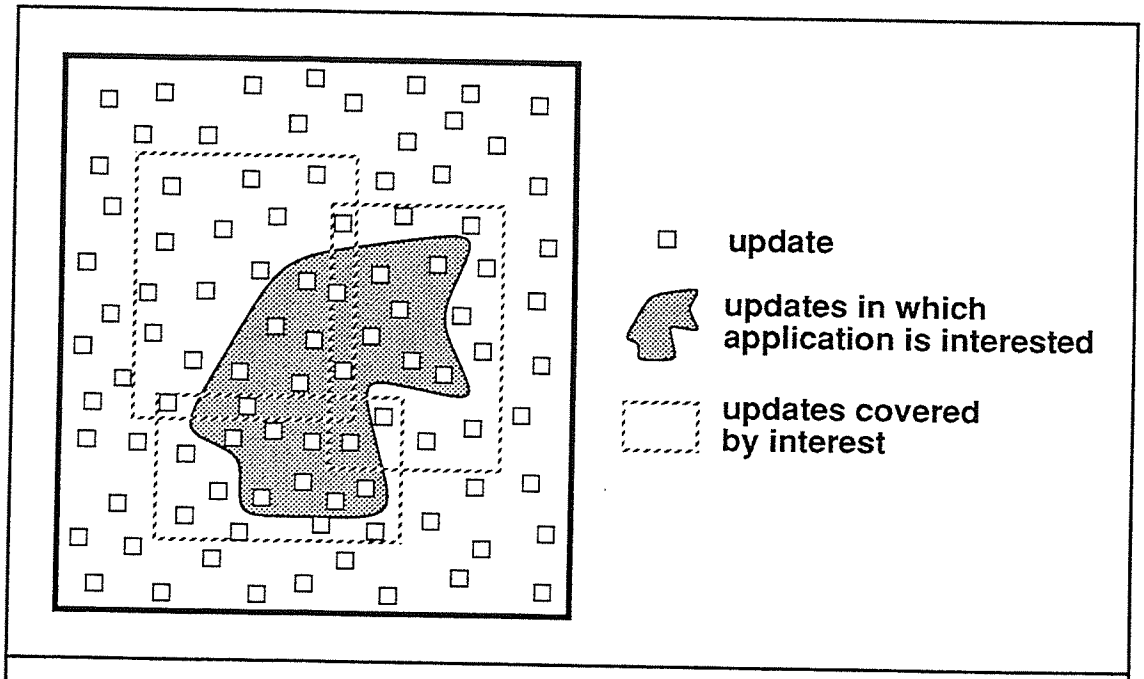

Figure 30: Interests.
Each interest conveniently specifies some set of updates. In this figure the application registers three interests with the interest matcher. These three interests include all the updates in which the application is interested (twenty-two in this case).

Collectively, a set of interests demarcates a **region of interest**. The region of interest should include updates to design data upon which the application is basing its operation—the "read set". When an application performs an update to the object cache, the interest matcher doesn't send a notification of that update back to the application. That update may, howev-

er, trigger the void propagator or the derived slot calculator to perform further updates on computed or derived slots, respectively; these updates may cause messages to be sent to the application that performed the original update if it has registered an interest in some computed or derived slot affected.

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

The mechanism of interests and messages provided by the TCM and DCM is what the Change Manager offers in place of exclusive locking, which is traditionally used, to maintain consistency between tools and the database. The TCM and DCM do use warning locks, which are free of deadlocks, to make the design data **active** [Dayal 88].

Matching Updates to Tool Interests

Table 4 below 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.

Incremental Computations

The ability to register interests and be informed of some subset of updates to the object cache is useful to an application that can perform incremental constraint checking or incremental recomputation of computed slots.

An application wishing to do so registers a region of interest which includes all updates to design data upon which the application's computations depend. The interest matcher ensures that the application is made aware of all updates in its region of interest. The application then performs incremental adjustments to its internal state and to the computed slot whenever messages describing updates in its region of interest arrive.

Table 4: Interests and Matching Updates

interest	**updates to annotated object cache that match interest, where updates may be performed by void propagator, derived slot calculator, or update notification handler**
value of slot S	S is a primitive slot: ChangeValue(S, v) S contains a subobject: application would use an "existence of object" or a "state of object" interest described below S is set-valued: CreateMember(S, X), DestroyMember(S, X), UndestroyMember(S, X), or any update to a member of S S is a reference slot: ChangeReference(S, OID) S is a computed slot: MarkVoid(S) or MarkValid(S) S is a derived slot: the derived slot calculator updates the value of the derived slot
existence of object X	DestroyMember(S, X), where X is a member of set-valued slot S DestroyMember(S, Y), where X is a subobject of Y UndestroyMember(S, X), where X is a member of set-valued slot S UndestroyMember(Y), where X is a subobject of Y
state of object X	any update to any slot of object X any update to any slot of a subobject of X

Incremental recomputation, when possible, permits computed slots to stay current without the computational expense of a complete recomputation [Overmars 83]. For example, there exist algorithms to perform incremental design-rule checking on integrated circuit layouts whereby the engineer remains aware of all outstanding design-rule violations [Ousterhout 83]. Note that the use of differential update by tools, as explained in Section 3.B.4, is required in order to make incremental recomputation possible.

If an application recomputes a computed slot immediately whenever the slot is voided, 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 also can achieve demand-driven computation by deferring recomputation of a computed slot until the value of that slot is needed.

The value of computed slots, such as constraints, is useful information to an engineer because it will affect his personal design methodology and planning, such as invoking constraint-checking mechanisms or embarking on a particular subgoal such as updating one aspect of the design.

If CPU resources were infinite, the value of computed 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 computed slots valid, and thus the choice between use of data- or demand-driven computation, is an engineering tradeoff.

Programmatic Interface

RegisterInterest(SpecificationOfInterest)
returns InterestID

An application enlarges its region of interest by calling RegisterInterest and passing the specification of an interest to be registered. When an update occurs to the object cache that matches an interest that the application has registered, the interest matcher sends a message to the application.

UnregisterInterest(InterestID)

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

Section 5.B.7: Locate Updated Dependencies of Computed Slots

When an application needs to recompute the value of a void computed slot, it may be useful to know which slots have changed since the slot was last valid. The **dependency locator** in the TCM compares timestamps in the annotated object cache to locate those slots.

When a design object is cached in a tool, it contains timestamp annotations from the DDMS clock. When the update notification handler merges updates into the annotated object cache, it assigns the timestamps of the notifications to timestamp annotations of cached objects; these timestamps are also from the DDMS clock. When an application makes an update to the annotated object cache, however, the timestamps used are from the tool clock. Thus timestamps in the annotated object cache will be a mix of timestamps from the DDMS clock and from the tool clock. Suppose t_1 and t_2 are two timestamps. We define a total ordering on timestamps as follows:

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

t_1 and t_2 are DDMS timestamps and t_1 is less than t_2
t_1 and t_2 are tool timestamps and t_1 is less than t_2
t_1 is a DDMS timestamp and t_2 is a tool timestamp

As explained in Section 4.B.5, each slot in an object is annotated by a timestamp. Updates to the annotated object cache performed by the application, void propagator, derived slot calculator, and update notification handler all maintain the invariant that if a computed slot C is void, its value depends upon slot S (as defined by the schema), and S has been updated since C was last valid, then $timestamp_C \leq timestamp_S$.

The dependency locator works by comparing the timestamp of the computed slot C with the timestamp of each slot S upon which it depends. If the timestamp of C is no greater than the timestamp of S, then S is included among those slots which, as a result of being updated, caused C to become void.

Programmatic Interface

LocateUpdatedDependencies(OID, ComputedSlot)
returns set Slot

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

Section 5.B.8: Commit Updates to Workspace

When an application wishes to save its updates of design objects to the workspace it has selected, it **commits** to the DDMS. When the application commits its update, the TCM 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 to the DDMS by calling UpdateWorkspace, as discussed in Section 4.B.9. An application may request that its updates be discarded rather than committed; in this case the TCM reloads cached objects from the database and reinitializes the annotated object cache. Table 5 below shows how the object cache manager in the TCM computes the update delta by recursively scanning each design object in the annotated object cache.

Table 5: Computing the Update Delta

type of slot S in object X	**the state of annotations to slot S**	**update generated, where t_{commit} = DCM time of commit by tool**
primitive	*value status* = different	*value status* ← same *timestamp* ← *timestamp* + t_{commit} The timestamp of each slot in the object cache that was updated is incremented by the current DDMS time when the tool commits, in order to convert it from tool time to DDMS time. ChangeValue(X, S, *value*, *timestamp*)

Table 5: Computing the Update Delta

type of slot S in object X	**the state of annotations to slot S**	**update generated, where t_{commit} = DCM time of commit by tool**
subobject or member of set	*value status* = different	*value status* ← same *timestamp* ← *timestamp* + t_{commit} Recursively scan each slot in object and generate updates, as explained in this table.
set-valued	Generate updates for each member where: (*existence status* = not in workspace; created)	*existence status* ← created in workspace; unchanged *timestamp* ← *timestamp* + t_{commit} CreateMember(X, S, OID of new member, *timestamp*)
	Generate updates for each member where: (*existence status* = created in workspace; destroyed)	*existence status* ← destroyed in workspace; unchanged *timestamp* ← *timestamp* + t_{commit} DestroyMember(X, S, OID of member, *timestamp*)
	Generate updates for each member where: (*existence status* = destroyed in workspace; undestroyed)	*existence status* ← created in workspace; unchanged *timestamp* ← *timestamp* + t_{commit} UndestroyMember(X, S, OID of member, *timestamp*)
object reference	*value status* = different	*value status* ← same *timestamp* ← *timestamp* + t_{commit} ChangeReference(X, S, *value*, *timestamp*)
computed	*validity status* = **void** and *voided* = **true**	*voided* ← **false** *validated* ← **false** *timestamp* ← *timestamp* + t_{commit} MarkVoid(X, S, *timestamp*)
	validity status = **valid** and *validated* = **true**	*voided* ← **false** *validated* ← **false** *timestamp* ← *timestamp* + t_{commit} MarkValid(X, S, *timestamp*) to clear the old value of slot S Recursively scan the value of the computed slot and generate updates according to this table. MarkValid(X, S, *timestamp*)

Table 5: Computing the Update Delta

type of slot S in object X	the state of annotations to slot S	update generated, where t_{commit} = DCM time of commit by tool
derived	*value status* = different	*value status* ← same *timestamp* ← *timestamp* + t_{commit} Recursively scan the value of the derived slot and generate updates according to this table.

Programmatic Interface

CommitUpdates(LastMessageHandled)
returns ok / handleMessages / invalidConstraint

An application commits its updates by calling CommitUpdates. The request will fail either if the application has not handled all the messages sent to it by the interest matcher, or if not all constraint requirements of the workspace selected by the application are **true** in the object cache.

AbortUpdates(LastMessageHandled)
returns ok / handleMessages

An application discards its updates by calling AbortUpdates. The request will fail if the application has not handled all the messages sent to it by the interest matcher.

Part 5.C: Invariants Maintained by the Tool Change Manager

All modules within the Tool Change Manager work together to jointly provide a collection of services to an application. Guaranteeing internal consistency and correct operation of the TCM requires that it maintain a number of invariants. These invariants motivate the details presented in the tables in Part 5.B of this chapter; this part of the chapter summarizes the invariants.

Section 5.C.1: Object Cache Related to Workspace by Update Delta

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

Invariant #1:

Let $V_W(t)$ be the view of design data at time t in the workspace selected by a tool and $V_T(t)$ be the view of design data in the tool's object cache at time t.

Then $V_T(t) = V_W(t) + \Delta(t)$, where Δ is the update delta which represents the uncommitted updates to objects in the cache. The update delta is a synthesis of updates to the object cache by the application and updates to the workspace from other tools.

The object cache manager preserves this invariant by incorporating updates from the application into the object cache. The update notification handler preserves this invariant by merging updates from other tools, as described by update notifications, into the object cache and making adjustments to the update delta; the way this is done is explained in Section 5.B.5.

Section 5.C.2: Annotations Provide Sufficient Information for Commit

An application can choose to commit its updates to the DDMS at any time. When it does so, the object cache manager scans the annotated object cache, interprets the annotation to determine what updates were performed, and generates a list of updates which are then presented to the DDMS.

The invariant which makes this possible is:

Invariant #2:

The update delta can be computed at any time from the annotations in the object cache.

The object cache manager, void propagator, update notification handler, and derived slot calculator are the only modules that update the annotated object cache. They alter the annotations in such a way that the difference between the state of objects in the cache and objects in the DDMS is captured by the state of the annotations. Refer to Tables 1, 2, 3, and 5 for details.

Section 5.C.3: Unhandled Messages Restrict Updates by Application

When an application makes updates to design objects in the object cache, it does so based upon the state of the objects it had read earlier. If the application has registered interests in certain updates, it may receive message from the interest matcher. The TCM guarantees:

Invariant #3:

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

This invariant 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. The protocol is detailed in Section 5.B.4.

Note that this invariant does not guarantee that the application has responded in a proper fashion to the messages it has received. Such a guarantee of the application's behavior cannot, in general, be enforced by the TCM.

Section 5.C.4: Computed Slots are Automatically Voided

An application need not be be aware of all computed slots which may be affected by an update, because the void propagator guarantees:

Invariant #4:

For every slot S and computed slot C that depends upon S:
If S is updated, then the *validity status* of C is set to **void**.

The void propagator behaves as a simple truth maintenance system by recursively marking computed slots as **void** after each update to the object cache made by either the application or by the update notification handler. Void propagation is discussed in Section 5.B.4.

Section 5.C.5: Relative Timestamps of Slots are Maintained

The dependency locator determines for a specified computed slot which slots upon which it depends have been updated since the computed slot was last computed. In order to do so, the dependency locator compares the timestamp of the computed slot with the timestamps of the source slots; if a computed slot has a timestamp no later than that of a source slot, then the source slot may have been updated since the computed slot was last computed and should be included in the answer returned by the dependency locator.

The invariant which guarantees that the algorithm used by the dependency locator works is:

Invariant #5:

For every slot S and computed slot C that depends upon S:
if S has been updated since C was last computed
then $timestamp(C) \leq timestamp(S)$

This invariant is maintained by the void propagator as follows:

When the void propagator is voiding a computed slot, and the computed slot is **valid**, the void propagator marks the slot as **void** and sets the timestamp of the computed slot to that of the update. If the computed slot has already been marked **void**, the void propagator changes the timestamp of the slot to be the minimum of the timestamp of the update and the timestamp already assigned to the computed slot.

Chapter 6

CAD Applications for Concurrent Design

The preceding two chapters have provided an operational definition of the DDMS Change Manager and the Tool Change Manager, and have identified invariants that the operation of the DCM and TCM guarantees. Despite the inclusion of the Change Manager within tools and the DDMS, and the limitation that an application accesses design data through the TCM, a tool may fail to operate suitably in an environment for concurrent design. This is because there are certain requirements on the application code within the tool to make it "well-behaved". Among well-behaved tools there is a range of levels of coordination of the application code with the TCM; higher degrees of coordination admit higher levels of concurrency within the design environment.

This chapter specifies what is required of application code for it to be well-behaved and what minimal alterations are needed to make an existing application well-behaved, explains what it means for an application to handle messages from the interest matcher, and discusses levels of coordination of the application code with the TCM. This chapter concludes with a list of characteristics of the "CAD tool of the future".

Part 6.A: The Well-Behaved Application

The Change Manager gives multiple applications simultaneous update access to design data. This places special requirements on applications so that they do not interfere with

each other. An application that meets the requirements is said to be **well-behaved**. This part of the chapter explains what those requirements are.

Section 6.A.1: Absence of Exclusive Locking

As explained in Chapter 3, the most significant benefit to the concurrent design environment offered the Change Manager is the use of asynchronous messages, rather than exclusive locking, for concurrency control. Exclusive locking is a syntactic method of ensuring that updates made by one application do not interfere with updates made by other concurrently-executing applications. As mentioned in Section 3.B.1, this technique is commonly used by traditional databases in business applications, which have only short-lived transactions. When an application acquires an exclusive lock on a design object, the application knows that the object will not be affected by updates made by other applications.

The cost of exclusive locking is the prohibition of access of data by one application while another application is updating the same data. This is not a high cost if the duration of access and update is short. In a design environment, however, design activities are long-lived, and a prohibition of concurrent access severely restricts concurrent design.

When the Change Manager, along with its asynchronous messages, is used, design objects checked-out by one application are subject to change, due to updates by other applications. The Change Manager does offer two important guarantees:

1. The update notification handler will incorporate updates made by other applications into the object cache.
2. The interest matcher delivers asynchronous messages to an application to notify it of updates to the object cache which match interests registered by the application.

The Change Manager does not guarantee non-interference of applications, as does exclusive locking. The responsibility that an application not interfere with updates of other applications belongs to the application. The Change Manager only provides assurance that

notifications of possibly conflicting updates will be delivered to tools which are sharing updates to the same design objects.

Section 6.A.2: Definition of Well-Behaved

In order for an application to be well-behaved and not interfere with updates of other applications, it must meet the following three requirements:

1. Even though the application is linked with the TCM and the object cache within the TCM may be accessible within the same address space as the application, the application must access the object only through the programmatic interface provided by the TCM, as defined in Chapter 5.
2. As the application operates and reads data from the object cache and initializes internal data structures, it must adjust its region of interest to include updates to all values upon which it is currently basing its internal state, so that it will receive messages from the interest matcher when those values change because of updates by other applications.
3. The application must handle all messages sent to it by the interest matcher before it commits its updates to the DDMS. What "handle" means is covered in the next section.

Note that a tool can be well-behaved according to the above definition merely by never committing its updates! Thus a application's being well-behaved does not imply that it is *useful* to an engineer, but merely not injurious to other applications' updates. If an engineer uses an application which fails to follow the above requirements, unpredictable alterations to design objects in the DDMS may result; in such a case not only will progress on the design fail to be effected, but also damage to or reversal of the contributions of other engineers may occur.

Section 6.A.3: Conversion of Existing Applications to be Well-Behaved

As mentioned in Section 6.A.2, an application, although well-behaved, may fail to be useful. This section will discuss how an existing CAD application, which was not built to be

used in a concurrent design environment, can be converted to be well-behaved and be somewhat useful, but still not need to know how to handle any messages. Although such a conversion ensures that the application will not disturb the efforts of other applications, the application will be unable to effect progress on the design in the face of concurrent access to the design by other applications.

Here is what is required of the tool:

- Upon starting, the application registers itself with the TCM.
- The application checks-out for update any design objects it needs to change, and checks-out for read any other design objects it needs to access.
- The application initializes internal data structures based on data it reads from the object cache, and specifies to the interest matcher a region of interest that covers those data.
- The application will perform its task as usual, including interacting with the engineer 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 DDMS.
- 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 DDMS. But if any messages have arrived from the interest matcher, the application must inform the engineer using it that the updates must be aborted and the tool restarted.

The application must abort its updates if messages have been sent from the interest matcher. That is because a simple-minded tool that lacks the intelligence to handle messages must not predicate any updates to engineering data upon other data 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 design, but would be considered well-behaved and thus would not adversely affect updates by other applications. Note that this scheme is analogous to optimistic concurrency control: acquisition of the same lock by two

processes requires that one process abort, but if there is no such interference then both processes can commit their updates.

Part 6.B: Handling Messages

Even though the update notification handler has incorporated external updates into the tool's object cache, there remains a very difficult question: Has the application's internal state, that is, its view of the world that it built from the object cache *before* the update notifications were received, been disturbed? The answer is—maybe. It depends upon the semantics of the application; these are known only to the application itself. The best assistance that can be offered to the application is to let it tell the interest matcher what design objects and slots it has assumed to be static, then notify it via messages if any of those change.

The above protocol presupposes that in most cases a tool will be able to handle notifications it receives. This is a form of optimistic concurrency control. There is precedent to this approach of requiring clients of the database to adjust to shared updates: this was the only form of concurrency control offered by the CODASYL database system [CODASYL 71]. In the worst case a tool is unable to incorporate the changes made by another tool into its view and the engineer cannot continue his current thread of updates; this is analogous to a database transaction being aborted. But in practice, the motivation for permitting multiple tools to update shared design objects was to allow *cooperating* updates to be made. And cooperation means that updates made by one agent and his tools are not catastrophic to the ongoing efforts of another agent. This does not mean that the update might not cause problems with the functionality or correctness of the design. In general, a tool, perhaps under direction of its agent, will try to adjust to changes made by other tools so that overall functionality and correctness of the design are retained.

As described in Section 5.B.6, each application submits interests to the interest matcher which detail its region of interest. As a result, an update within that region of interest results in a message being sent to that application. As described above, in order to be well-behaved an application must handle every message received, if it is to commit its updates to the DDMS.

A message from the interest matcher is 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 DDMS. This might also include updating a graphical display which offers the engineer a view of the design data. (Whether the display should be automatically updated, or whether that would require approval from the engineer, depends upon the semantics of the tool and upon human factors which are beyond the scope of this research.) To handle a message may also mean that the application must perform compensating updates on the object cache in order to restore semantic invariants of the design data or to amend changes which were performed automatically by the update notification handler.

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

The requirement that applications respond in a reasonable fashion to messages is not trivial. In the general case, handling a message may require of the application an arbitrary amount of intelligence; converting existing applications to intelligently handle notifications is a dif-

ficult task. The amount of intelligence that an application has is called as the "level of coordination" of the application with the TCM, and is discussed in Part 6.C below.

Part 6.C: Application Coordination with the Tool Change Manager

In many cases an application will be able to handle a message from the interest matcher 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 not have enough intelligence to enable it to handle a particular message from the interest matcher.

The variety of circumstances that an application is able to handle messages from the interest matcher and still continue to operate without aborting the updates it has made is called the "level of coordination" of the application with the TCM.

An application that has achieved a low-level of coordination with the TCM may frequently be forced to abort operation when other applications update design data upon which it has based its internal state. An application with a high-level of coordination with the TCM can usually continue operating even when there are updates to shared design data that are performed by another applications.

Section 6.C.1: Low Level of Coordination

If an application lacks the intelligence needed to handle messages from the interest matcher, then it cannot commit its changes to the DDMS if another application updates design

data upon which it has based its internal state. Such a simple-minded tool might merely inform the engineer using it that the tool must be restarted.

This is a low-level of coordination of the application with the TCM. It is interesting to note that this corresponds to optimistic concurrency control, as described in Section 6.A.3.

Section 6.C.2: Medium Level of Coordination With Commutative Operations

Certain combinations of updates commute. For example, adding member m_1 then member m_2 to a set has the same result as adding them in the opposite order. An application which recognizes commutative operations on design objects can mechanically handle those messages from the interest matcher that specify 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 coordination better than the lowest level described in the preceding section: the more messages an application can handle, the less frequently it will be forced to abort operation and discard updates made by the engineer.

Section 6.C.3: High Level of Coordination

If an application keeps its region of interest current, and handles all messages received from the interest matcher, then it does become possible for that application to operate concurrently with other applications sharing the same design data and never need to abort because it does not understand an update made by another application. This is a high level of coordination of the application with the TCM.

An important premise of this thesis is that forced by a need for a high degree of concurrency, applications will evolve toward a high level of integration. This level of integration 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 engineer.

A high level of coordination 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 Change Manager 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 interest matcher.

There are two important benefits from a high level of integration:

1. Tools from multiple vendors can be operated concurrently without understanding each other's semantics. Instead, a tool needs to understand only the semantics of the view of the design which it accesses. This is called **semantic encapsulation.**
2. The designer is not forced into a specific order of tool invocation, as is the case when exclusive locking on design objects is used. The design process can instead be viewed as an evolution, rather than as a series of disconnected activities. A high level of coordination removes this temporal disjointedness, and enables engineers to use a design-centric rather than a tool-centric methodology.

Section 6.C.4: Other Levels of Coordination

There exist many levels of coordination between the low and high levels. It is not necessary for an application to have a high level of coordination with the TCM in order to obtain any benefit; it is simply that *a higher level of coordination derives greater benefit.* Thus coordination need not be a single expensive error-prone conversion, but rather a migration toward an increasingly intelligent and powerful set of design tools.

Section 6.C.5: Examples of Application Coordination

Layout Editor

Consider a tool T that allows engineers to place, move, and remove rectangles from a parent rectangle. A VLSI layout editor, for example, is a tool with similar functionality. Assume the application in T keeps its region of interest current with the interest matcher. Suppose two engineers E_1 and E_2 are collaborating and are jointly updating the same design D using instances T_1 and T_2 of tool T, and that E_1 commits his changes to the DDMS. This results in a number of notifications being sent from the DDMS to the update notification handler in T_2, which in turn will merge the updates from T_1 into the object cache and send messages to the application in tool T_2 that describe how the object cache has changed as a result of updates to D made by another tool (T_1 in this case). The application in T_2 might respond to the messages from the interest matcher in any of the following ways:

T is not well-behaved:

T_2 ignores messages from the interest matcher that describe updates made by T_1 and which have been merged into the object cache. Thus the view of D presented by T_2 to E_2 fails to show changes made by E_1. Engineer E_2 sees a view of design D that is incorrect, and will probably make erroneous updates to D based on that incorrect view. The result is that E_2 may unwittingly hinder progress on, or damage the consistency of, design D.

T is well-behaved and has been coordinated with the TCM at a low level:

At some time no later than when T_2 commits its updates to the DDMS, the application in tool T_2 notices that it received a message from the interest matcher, realizes that its read set may have been disturbed, and notifies E_2 that it cannot continue operation and that the updates that E_2 had made must be discarded and that the operation of T_2 must be aborted. (In rare cases a supervisor might have enough knowledge of the semantics of design data to permit E_2 to commit his updates despite updates made by E_1.)

As mentioned in Section 6.C.1, this is similar to optimistic concurrency control. This behavior maintains the consistency of the design, but in this case tool T is not as useful to E_2 as it might have been if T could intelligently handle messages from the interest matcher.

T is well-behaved and has been coordinated with the TCM at a high level:

When the application T_2 sees the messages from the interest matcher, it incorporates knowledge of those updates into its internal data structures. Tool T_2 furthermore adjusts the view of the design which is presented to E_2 to reflect those changes, so that E_2 can make decisions based upon the current state of design D, including updates made by E_1.

Bank Database

Suppose that the East Sandwich Bank on Cape Cod uses the Change Manager. The bank offers a savings account to each customer. Each customer is represented in the bank's database by an object C that has one slot S that represents the current balance in the savings account. The bank offers one tool which customers can use at automated teller machines: "MakeDeposit", which displays the current balance then accepts a single number to allow a customer to increment the balance of his savings account.

The Smythe family is a customer at ESB; all members of the family use the same savings account. Ellen and Nancy Smythe are fond of using two instances of MakeDeposit simultaneously on payday. Suppose the current balance is $100, and that Ellen and Nancy have $20 and $30, respectively, to deposit. Consider the following sequence of events:

Ellen starts an instance of MakeDeposit, say $MakeDeposit_E$. $MakeDeposit_E$ caches C_{Smythe}, places an interest on updates to $C_{Smythe}.S$, and displays $C_{Smythe}.S = 100$.

Nancy starts an instance of MakeDeposit, say $MakeDeposit_N$. $MakeDeposit_N$ caches C_{Smythe}, places an interest on updates to $C_{Smythe}.S$, and displays $C_{Smythe}.S = 100$.

Ellen tells $MakeDeposit_E$ to accept $20, places her deposit in the slot, and tells $MakeDeposit_E$ that she has finished. $MakeDeposit_E$ commits the update

$C_{Smythe}.S \leftarrow 120$ to the database and informs Ellen that her transaction has completed. This results in a message from the interest matcher to be sent to $MakeDeposit_N$ that $C_{Smythe}.S$ changed from 100 to 120.

Nancy tells $MakeDeposit_N$ to accept $30, places her deposit in the slot, and tells $MakeDeposit_N$ that she has finished.

$MakeDeposit_N$ might pursue any of three courses of action in this situation, depending upon whether MakeDeposit is well-behaved and at what level MakeDeposit is coordinated with the TCM:

MakeDeposit is not well-behaved:

$MakeDeposit_N$ ignores the message from the interest matcher, commits the update $C_{Smythe}.S \leftarrow 130$ to the database, and informs Nancy that her transaction has completed. Note that $20 has been lost, and that the consistency of the bank's database has been disturbed. In this situation the programmer responsible for writing MakeDeposit should be fired! The need for applications to be well-behaved should now be apparent.

MakeDeposit is well-behaved and has been coordinated with the TCM at a low level:

When Nancy asks $MakeDeposit_N$ to commit her deposit, $MakeDeposit_N$ notices that it received a message from the interest matcher, realizes that its read set has been disturbed, and notifies Nancy that her transaction has been aborted.

As mentioned in Section 6.C.1, this is similar to optimistic concurrency control. This behavior maintains the consistency of the database, but MakeDeposit is not as useful to Nancy as it would be if it had a higher level of coordination.

MakeDeposit is well-behaved and has been coordinated with the TCM at a high level:

When Nancy asks $MakeDeposit_N$ to commit her deposit, $MakeDeposit_N$ notices that it received a message from the interest matcher, sees that the value of $C_{Smythe}.S$, which it had thought was 100, is instead 120, and incorporates knowledge of the change into the update it now submits to the database—$C_{Smythe}.S \leftarrow 150$—and informs Nancy that her transaction has completed.

This behavior both maintains the consistency of the database and enables Nancy's request to be satisfied. A high level of integration in this case does require that MakeDeposit$_N$ understand the semantics of making a deposit and automatically restart itself, or incorporate the new value of C_{Smythe}.S in its calculations, rather than abort.

Part 6.D: The "CAD Application of the Future"

The CAD application of the future, which when linked with the TCM becomes a CAD tool useful in the concurrent design environment, has characteristics which differ from those of traditional applications:

- It doesn't access design data in the design database directly but instead manipulates cached copies of design objects through a well-defined programmatic interface.
- It adjusts its region of interest to include updates to those data 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 obligated to handle them before committing updates from the annotated object cache to the DDMS.
- It uses the design status monitor to stay aware of the design status and makes the design status known to the engineer using the tool.
- It gives the engineer using the tool a means to log a conflict in order to identify updates by another engineer as unacceptable.

Chapter 7

Conclusions and Future Work

Part 7.A: Key Concepts

Section 7.A.1: Concurrent Design is Required for Complex Designs

Many engineering designs are now so complex that the effort of a team of engineers is required if the design is to be completed in a reasonable amount of time. This approach is known as *concurrent design.*

Concurrent design is enhanced if engineers and their tools are able to cooperate at a range of levels during the course of the design process. An important advantage of close cooperation is that conflicts can be identified early in the design process and thus reduce expensive redesign [Boehm 81].

Section 7.A.2: Traditional Approach is Inadequate

This section summarizes the ways in which traditional databases and tools are inadequate for use in a design-centric environment. A new approach which corrects these weaknesses—the Change Manager—is the subject of this thesis.

An Alternative to Exclusive Locking Must Be Developed

Exclusive locking on design objects is too restrictive, and severely limits close cooperation in a design environment. The use of exclusive locks prevents tools from accessing design

objects if any other tool is updating those objects. Methods other than exclusive locking must be used by the database to ensure consistency between the cache of design data within tools and the design database.

A Workspace Hierarchy is Needed

There are times when updates of a tentative or experimental nature need to be performed without affecting the existing state of objects. This situation may arise repeatedly, and requires that the design database support a hierarchy of workspaces, in which workspaces closer to the root workspace hold designs which are more stable. The next-generation design database must offer a workspace hierarchy and associated protocol of check-out and check-in that accommodates the hierarchy. These processes are detailed in Sections 4.B.2 and 4.B.6.

Support for Cooperation Is Absent

Traditional databases assume that users do not cooperate informally. Hence they do not give support for concurrent operation. The next-generation design environment will permit access to design objects without excluding access to those objects by other engineers and tools. There may be times, however, when engineers fail to coordinate or make erroneous updates.

The next-generation design environment, which will support cooperation, must model the concept of **conflict** in which one engineer's efforts have been adversely affected by those of another engineer. Knowledge of conflicts can be used to initiate recovery from high-level concurrency failures which are are identified by agents and intelligent tools. The supervisor has responsibility of ensuring that all conflicts are eventually resolved. Section 4.B.10 discusses conflicts.

Engineers and Their Supervisors Need Access to Design Status

Information that might be considered "internal" in a traditional database, such as knowledge of which process is accessing what data, is useful to clients of the database in a design environment. Such process information is called **design status**. Design status can be used to initiate direct communication among engineers, to help plan and coordinate design activities, and to locate the source of updates which some engineer thinks to be incorrect or conflicting with efforts in progress.

The next-generation design database must give tools access to the design status, and give them the ability to track changes to the design status, that is, it must make the design status **active**. This is explained in Section 4.B.11.

Translators Must Be Part of the Tool

The next generation of CAD tool is responsible for translating the data model offered by the design database directly, or indirectly through mediators [Wiederhold 91], to and from its internal data structures.

Engineering Data are Dynamic

Traditional tools assume that the design data they are accessing are not updated by other tools. When an application assumes that design data it accesses are static, then that application is unable to operate concurrently with other tools that might access the same design data. But this is contrary to the nature of design-centricity. Thus the next-generation application must see engineering data as dynamic and subject to change during a design transaction. This means that an application must respond to updates made by other tools by making its internal state consistent with the new state of the database.

Constraint Invalidation is Haphazard

The degree of consistency of a design, modeled as a number of *constraints*, may be affected by updates to the design. In the traditional design environment, the marking constraints as invalid is done manually or in a haphazard fashion. The next generation of tool should automatically invalidate constraints which may have been affected by updates performed by the tool. This will ensure that designs are not given a higher degree of validation than they have actually achieved. Consistency checking of designs must be the responsibility of tools rather than the database, so that tool developers have the freedom to invent new ways of performing consistency checks on design data.

Section 7.A.3: Change Manager Provides Framework

A central mechanism or **framework** is needed that will send messages to applications so that they stay aware of updates to shared design data. The **Change Manager** provides such a framework. Tools may be adapted to use the Change Manager; this will enable them to work in a concurrent design environment. The Change Manager monitors updates to shared data and sends messages to tools whenever updates in their regions of interest occur.

The Change Manager consists of two collections of modules: the Tool Change Manager (TCM), which is part of a CAD tool, and the DDMS Change Manager (DCM), which is part of the Design Data Management System. The Change Manager supports *tool independence*: applications which make use of the Change Manager need not understand the semantics of other applications. This enables the Change Manager to be open-ended, whereby neither it, nor the tools which make use of it, need be changed when a new tool is introduced into the design environment.

Section 7.A.4: The DCM Adds Capability to an Object Store

Since traditional databases are inadequate for use in a concurrent design environment, additional techniques are needed. The DDMS Change Manager (DCM) adds capabilities to an object store in order to make it suitable for use as a Design Data Management System. These capabilities are described in detail in Chapter 4 and are summarized below:

- tool registration

  Tools can register the commencement and termination of their operation with the DCM.

- design object check-out and check-in

  Tools can check-out design objects for update or for read access, and check-in objects they have checked-out when access is no longer required.

- asynchronous update notifications

  The DCM sends asynchronous update notifications to tools. These notifications describe updates made by other tools to design objects which have been checked-out.

- workspace hierarchy and constraint requirements

  The DCM offers a hierarchy of workspaces into which updates may be encapsulated, and enforces constraint requirements.

- active design status

  The DCM gives tools access to the design status and sends notifications to tools when the design status changes.

- conflict logging

  The DCM gives tools the ability to mark updates by other tools as conflicts, and ensures that a workspace cannot commit to its superior workspace if it contains unresolved conflicts.

- resolution of race condition between a tool and the DDMS

  The protocol of requests and acknowledgements between a tool and the DDMS prevents tools from making updates to the DDMS with incomplete knowledge, even in the face of delays in the transmission of update notifications.

Section 7.A.5: The TCM Offers Services to an Application

The Tool Change Manager (TCM) offers services to a CAD application which simplify the development of tools that can operate effectively in a concurrent design environment. These services are described in detail in Chapter 5 and are summarized below:

- consistency of the object cache within a tool

  The TCM keeps the object cache consistent in the face of both internal updates made by the application and external updates performed by other tools.

- automatic voiding of computed slots

  The void propagator in the TCM uses the *computedSlotSpec*s of the schema to automatically void computed slots when slots upon which they depend change. This causes constraints throughout a design hierarchy to be marked as invalid when low-level components are changed. Thus applications need not be aware of all constraints and other computed slots in which design data participate, nor of the manner in which those slots depend upon the values of other slots of design objects.

- interests and messages

  The TCM will monitor changes to data upon which an application has registered an interest, and will inform the application when such an update occurs. The application can use this information to make its internal data structures consistent with the object cache.

  An application which performs consistency checks or recalculates computed slots can use notifications to incrementally revalidate the constraint or computed slot. This can result in a tremendous reduction in the computation required.

- deferred handling of update notifications

  At times, an engineer may wish to ignore updates which are made by tools other than the one that engineer is using. For this reason, the TCM gives the application a programmatic interface to defer the incorporation of external updates into the object cache.

Part 7.B: Contributions

This part of the chapter identifies the research contributions of this thesis.

Section 7.B.1: Flexible Model of Concurrency Control

The use of a traditional database in the design environment means that exclusive locking is used to maintain consistency of design data. Exclusive locking severely restricts engineers' use of tools to work on designs. An engineer must carefully plan design activities so that they do not preclude access to design data needed by other members of the design team. This tool-centric approach is inappropriate in the concurrent design environment, where engineers communicate and coordinate their updates in order to achieve progress on the design.

The most significant contribution of this thesis is the introduction of a flexible model of concurrency control, which does not necessitate the use of exclusive locks, into the design environment. The absence of exclusive locks makes a high degree of cooperation possible among a team of engineers who are collaboratively completing a design. Informal collaboration (directly among the engineers) as well as formal collaboration (computer-mediated) is supported.

Use of Asynchronous Notifications in Place of Traditional Exclusive Locking

The Update Monitor in the DDMS Change Manager provides the mechanism through which a tool can become aware of updates made by other tools to cached design objects.

The DCM sends asynchronous update notifications to the tools which describe updates that occur.

Applications Handle Notifications

When applications use this mechanism and follow the requirements to make them "well-behaved", as explained in Section 6.A.2, they can keep their internal data structures consistent with be the state of design objects in the DDMS without the need for restrictive exclusive locks.

Since design data are dynamic, and access to them is shared during collaboration, particularly in situations where there is a high degree of cooperation among engineers, the design applications of the future must respond flexibly to update notifications from the DCM. The update notification handler in the TCM automatically incorporates updates from other tools into the object cache, but the application has the responsibility of updating its internal data structure.

A Range of Levels of Coordination is Possible

Applications can be coordinated with the Tool Change Manager (TCM) in varying degrees. A range of possible techniques is possible, all of which guarantee consistency. Degrees of coordination differ in the amount of tool-specific knowledge required. Use of knowledge offers a **high level of coordination** with the TCM and enables a tool to respond flexibility to update notifications rather than abort operation. A suite of tools which are coordinated with the TCM at a high level permits engineers to concurrently share updates to design objects. Section 6.C.3 discusses coordination of applications with the TCM.

Section 7.B.2: Object Model for a Design Database

Chapter 2 presented 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 presented

in this thesis is a formal model, and was used in Chapters 4 and 5 to explain operation of the Change Manager. Section 7.C.2 discusses extensions to the object model.

Section 7.B.3: Annotations on Objects

Section 4.B.5 described auxiliary information attached to design objects, called **annotations**, that represent the difference between the state of objects in a workspace and the state of those objects in its superior workspace. The DDMS uses this information to compute the update delta when a workspace is to be committed to its superior workspace.

The same annotations enable a tool to track how design objects in its cache differ from those in the database. The tool uses this information to compute the update delta when it needs to commit its updates to the database.

Section 7.B.4: Handshaking Protocol Between Tool and DDMS

The Update Monitor in the DCM uses an optimistic handshaking protocol in which a tool cannot commit changes unless it has processed all pending notifications. This protocol resolves the race condition between a tool and the DDMS which may result from a delay in the transmission of an update notification from the DCM to the TCM. The protocol is presented in Section 4.B.9.

Section 7.B.5: Architecture of the Change Manager

Chapters 4 and 5 present the architecture and operation of the Change Manager. The Change Manager is distributed among the DDMS and tools, as the DDMS Change Manager and the Tool Change Manager, respectively. The DCM and TCM jointly provide the features required in a design-centric design environment, as explained in Part 3.B, and satisfy the requirements presented in Section 3.C.2. The Change Manager provides the *mechanism* with which many *policies* of design methodology can be implemented.

Part 7.C: Future Work

This part of the chapter describes four directions for future work on the Change Manager.

Section 7.C.1: Prototype of Change Manager

Since the notion of a Change Manager is new, there do not yet exist next-generation tools or databases that use implementations of the Change Manager. The construction of prototype Change Manager, DDMS, and tools is an important aspect of future work. Working prototypes will demonstrate the feasibility and utility of the Change Manager approach to design environments.

Some of the concepts that a prototype will demonstrate are listed below:

- Changes are propagated among tools which share updates to the same design objects. For example, one tool might be a schematic capture program and the other a simulator, both making use of the same data.
- A tool receives notification of the fact that another tool has checked out the same design object for update. The tool can use this information to make the engineer using the tool aware of the potential for conflict.
- Constraints are automatically invalidated by the void propagator when data affecting the constraints are changed. (The state of being invalid is not committed to the DDMS, however, until the associated updates are.)
- Requests to commit workspaces are allowed to proceed only if all constraint requirements have been validated.

Status of Prototype

Some prototyping of elements of the Change Manager has been performed on a Sun Sparcstation. The X Window system is being used for the user interface portion of the tools, Lisp/CLOS for the implementation language, and CLX for the bridge between Lisp and X. The

DDMS and tools each have a corresponding lightweight process so that they can operate asynchronously. The prototype has validated the approach of using a central mechanism (the DCM) to keep tools aware of asynchronous updates to design objects.

Section 7.C.2: Extensions to the Object Model

The object model defined in this thesis offers simple abstractions such as object, slot, value, set, ownership, and reference. Providing additional data structures and inter-object relationships in the object model would simplify the job of the application developer by providing additional abstractions with which design data could be organized and manipulated. This section discusses four such extensions to the object model.

Arrays

One data structure offered by many programming languages is that of **array** [Kernighan 88]. An array is a homogeneous collection of N objects, where N is the fixed size of the array. The objects in the array are updated individually, and are referred to by a numerical index. Interests and update notifications for arrays could distinguish among changes to any part of the array, changes to some subarray, and changes to an individual element.

Lists

Another useful data structure is the **list** [Aho 83][McCarthy 60]. A list is a sequence of objects. Objects can be inserted into or removed from a list. The number of objects in a list can vary, and the positions of the list's elements can shift as objects are inserted and deleted. Describing updates to particular elements or portions of a list in update notifications will require a common specification which does not depend upon the positions of the elements.

Automatic Evaluation of Simple Functions

The values of derived slots of objects are automatically recomputed when a source object changes. The object model can be extended to include operations other than equality and projection, such as arithmetic and logical operations, among those computations automatically performed. Which operations to include will require an analysis of engineering tradeoffs.

Section 7.C.3: Use of Domain-Specific Knowledge

The semantics of a particular engineering domain, e.g., of electronic design, can be used to develop views, interests, constraints, and methods which incorporate knowledge of that domain. Modules which offered domain-specific capabilities could, like the TCM, be included with each application and would simplify the task of the application developer. This section describes three examples of domain-specific extensions.

Multiple Views of a Design

It is not uncommon for two views of a design, such as the schematic diagram and parts list of a circuit board, to have non-isomorphic decompositions [Rubin 87]. There is nothing that would prevent an application from representing this with the existing object model, but there is now no explicit support for maintaining multiple non-isomorphic views of a design.

One extension to the Change Manager would be the automatic linkage of nodes in non-isomorphic hierarchies to the extent that they are isomorphic [Beetem 81]. The more general problem of determining how an update to one view affects other views is known as the **view update problem** and has been solved only in special cases [Keller 85]. Furthermore, use of the object model adds complexity, since, unlike the relational model in which views are derived from a set of normalized relations, there exists no foundation in the object model from which all object views can be derived.

Change Abstractions

Change abstractions are high-level interests which are translated from a domain-specific level to a sets of more primitive interests. An example of a change abstraction is "switching speed", which is translated to interests involving capacitance and resistance, and ultimately to interests in the values of slots in objects, such as length, width, and type of material.

Change abstractions are useful for two reasons:

1. The bandwidth of communication between the TCM and the application is reduced substantially. Not only can one change abstraction be mapped to many updates, but at times many updates can result in one high-level notification being sent to the application [Wiederhold 86b][Risch 89].
2. Change abstractions provide a programmer who develops applications with a more powerful vocabulary with which to talk about updates on designs, and thus reduce the complexity which the programmer must handle.

Pruning of Message Queue

When an update performed by a tool matches an interest registered by an application, a message describing the update is sent to the application. While an application is busy with calculations involving its internal data structures, a number of messages from the interest matcher may be queued for later processing.

One way to extend the interest matcher is to enable it to recognize messages which represent inverse and idempotent operations and prune the queue appropriately. For example, if a slot's value is set twice, or an object is destroyed then undestroyed, then the application need not receive messages which describe each update. Pruning of the message queue has the potential of significantly reducing the number of messages which the application has to handle.

Intelligent reordering of messages in the queue in a domain-specific manner is another way to improve the efficiency of a tool's operation. For example, sorting messages based upon the geometry of the update in a chip design would benefit a layout editor performing incremental design-rule checking, since it would keep all messages together which describe updates to a particular region.

References

[Aho 83] Alfred V. Aho et al. *Data Structures and Algorithms*, pages 37-52, Addison-Wesley, 1983.

[Backus 60] John Backus. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference", *Information Processing*, pages 125-132, 1960.

[Batory 85] D.S. Batory and Won Kim. "Modeling Concepts for VLSI CAD Objects", *ACM Transactions on Database Systems*, pages 322-346, September 1985.

[Beetem 81] John F. Beetem. "Structured Design of Electronic Systems Using Isomorphic Multiple Representations", Ph.D. thesis, Integrated Circuits Laboratory, Stanford University, December 1981.

[Bhateja 87] Rajiv Bhateja and Randy H. Katz. "Valkyrie: A Validation Subsystem of a Version Server for Computer-Aided Design Data", *Proceedings of the Twenty-fourth ACM/IEEE Design Automation Conference*, June 1987.

[Björnerstedt 88] Anders Björnerstedt and Christer Hultén. "Version Control in an Object-Oriented Architecture", Technical Report SYSLAB 57, Department of Computer Sciences, Chalmers University of Technology, May 1988.

[Boehm 81] Barry W. Boehm. *Software Engineering Economics*, Prentice-Hall, 1981.

[Breitbard 68] Gary Y. Breitbard and Gio Wiederhold. "PL/ACME: An Incremental Compiler for a Subset of PL/1", *Proceedings of the 1968 IFIPS Conference*, pages 358-363, 1968.

[Brodie 84] Michael L. Brodie et al. "CAD/CAM Database Management", *Database Engineering*, pages 64-72, June 1984.

[Bushnell 86] Michael L. Bushnell and S.W. Director. "VLSI CAD Tool Integration Using the Ulysses Environment", *Proceedings of the Twenty-third ACM/IEEE Design Automation Conference*, pages 55-61, June 1986.

[Ceri 84] Stefano Ceri. *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.

[CODASYL 71] *CODASYL Data Base Task Group Report*, ACM, April 1971.

[Dayal 88] Umeshwar Dayal et al. "Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System", *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, pages 129-143, September 1988.

[Dayal 90] Umeshwar Dayal et al. "Organizing Long-Running Activities with Triggers and Transactions", *Proceedings of the ACM SIGMOD Conference*, pages 204-214, May 1990.

[Doyle 79] Jon Doyle. "A Truth Maintenance System", *Artificial Intelligence*, pages 231-272, November 1979.

[Du 87] H. C. Du and S. Ghanta. "A Framework for Efficient IC/VLSI CAD Databases", *Proceedings of the Third International Conference on Data Engineering*, pages 619-625, February 1987.

[Eastman 91] Charles M. Eastman et al. "A Data Model for Design Databases", Technical Report 12, University of California at Los Angeles, March 1991.

[Ellis 89] C.A. Ellis et al. "Concurrency Control in Groupware Systems", *Proceedings of ACM SIGMOD*, pages 399-407, May 1989.

[Ellis 91] C.A. Ellis et al. "Groupware—Some Issues and Experiences", *Communications of the ACM*, pages 38-58, January 1991.

[Garcia-Molina 87] Hector Garcia-Molina and K. Salem. "Sagas", *Proceedings of the ACM SIGMOD Conference*, May 1987.

[Gray 81] Jim Gray. "The Transaction Concept: Virtues and Limitations", *Readings in Database Systems*, pages 140-150, Morgan Kaufmann Publishers, 1988.

[Hall 88] Keith Hall. "An Introduction to the Problems of Engineering Information Systems", *Proceedings of the IEEE Systems Design and Networks Conference*, pages 85-89, April 1988.

[Hall 89] Keith Hall et al. "Explicit and Implicit Change Coordination in an Information-Rich Design Environment", *Proceedings of the NSF Engineering Design Research Conference*, June 1989.

[Harrison 86] David S. Harrison et al. "Data Management and Graphics Editing in the Berkeley Design Environment", *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 24-27, 1986.

[Heiler 87] Sandra Heiler, Umeshwar Dayal, et al. "An Object-Oriented Approach to Data Management: Why Design Databases Need It", *Proceedings of the Twenty-fourth ACM/IEEE Design Automation Conference*, pages 335-340, June 1987.

[Henderson 89] Peter B. Henderson. "Integrated Design and Programming Environments", *IEEE Computer*, pages 12-16, November 1989.

[Katz 83] Randy H. Katz. "Managing the Chip Design Database", *IEEE Computer*, pages 26-35, December 1983.

[Katz 87] Randy H. Katz. "Managing Change in a Computer-Aided Design Database", Technical Report UCB/CSD 87/341, Computer Science Division, University of California at Berkeley, January 1987.

[Keller 85] Arthur M. Keller. "Updating Databases Through Views", Ph.D. thesis, Computer Science Department, Stanford University, February 1985.

[Kernighan 88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, Prentice Hall, 1988.

[Korth 86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*, McGraw Hill, 1986.

[Korth 90] Henry Korth et al. "A Formal Approach to Recovery by Compensating Transactions", *Proceedings of the Sixteenth Conference on Very Large Databases*, pages 95-106, August 1990.

[Lamport 78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, pages 558-565, July 1978.

[McCarthy 60] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM*, pages 184-195, April 1960.

[Mehmood 87] Z. Mehmood et al. "A Design Data Management System for CAD", *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 220-223, 1987.

[Moss 85] Eliot Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[Nodine 90] "Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications", *Proceedings of the Sixteenth Conference on Very Large Databases*, pages 83-94, August 1990.

[Ousterhout 83] John Ousterhout, Gordon Hamachi, et al. *A Collection of Papers on Magic*, Technical Report UCB/CSD 83/154, University of California at Berkeley, December 1983.

[Overmars 83] Mark H. Overmars. *The Design of Dynamic Data Structures*, Springer-Verlag, 1983.

[Papadimitriou 84] Christos H. Papadimitriou. "On Concurrency Control by Multiple Versions", *ACM Transactions on Database Systems*, pages 89-99, March 1984.

[Papadimitriou 86] Christos H. Papadimitriou. *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[Risch 89] Tore Risch. "Monitoring Database Objects", *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 445-453, August 1989.

[Risch 91] Tore Risch. "Building Adaptive Applications Using Active Mediators", *Proceedings of the Second International Conference on Databases and Expert Systems*, August 1991.

[Roussopoulos 91] Nick Roussopoulos et al. "An Architecture for High Performance Engineering Information Systems", *IEEE Transactions on Software Engineering*, pages 22-33, January 1991.

[Rubin 87] Steven M. Rubin. *Computer Aids for VLSI Design*, Addison-Wesley Publishing Company, 1987.

[Spooner 85] David L. Spooner et al. "Abstract Data Types for CAD Systems", *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 359-364, March 1985.

[Stefik 82] Mark Stefik et al. "The Partitioning of Concerns in Digital System Design", Technical Report HPP-82-2, Computer Science Department, Stanford University, February 1982.

[Ullman 82] Jeffrey D. Ullman. *Principles of Database Systems*, Computer Science Press, 1982.

[van der Meijs 85] N. van der Meijs et al. "A Data Management Interface to Facilitate CAD/IC Software Exchanges", *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1985.

[van der Wolf 88] Peter van der Wolf and Rene van Leuken. "Object Type Oriented Data Modeling for VLSI Data Management", *Proceedings of the Twenty-fifth ACM/IEEE Design Automation Conference*, June 1988.

[Wiederhold 80a] Gio Wiederhold and Ramez El-Masri. "The Structural Model for Database Design", *Entity-Relationships Approach to Systems Analysis and Design*, pages 237-257, North-Holland, 1980.

[Wiederhold 80b] Gio Wiederhold. "A Database Approach to Communication in VLSI Design", Technical Report 80-826, Computer Science Department, Stanford University, October 1980.

[Wiederhold 83] Gio Wiederhold. *Database Design*, McGraw-Hill Book Company, 1983.

[Wiederhold 85] Gio Wiederhold et al. "Models for Engineering Information Systems", *Proceedings of the 1985 VHSIC Conference*, December 1985.

[Wiederhold 86a] Gio Wiederhold and David Beech. "Object Management in Engineering Information Systems", unpublished manuscript, 9 pages, March 1986.

[Wiederhold 86b] Gio Wiederhold. "Support for Parallel Design in an Engineering Information System", unpublished manuscript, 20 pages, August 1986.

[Wiederhold 86c] Gio Wiederhold. "Views, Objects, and Databases", IEEE Computer, pages 37-44, December 1986.

[Wiederhold 88] Gio Wiederhold. "Engineering Information Systems: Prospects and Problems of Integration", *IEEE Spring COMPCON Digest of Papers*, pages 228-229, March 1988.

[Wiederhold 89] Gio Wiederhold et al. "Layering an Engineering Information System", *IEEE CS Spring COMPCON Digest of Papers*, pages 444-449, February 1989.

[Wiederhold 91] Gio Wiederhold. "Mediators in the Architecture of Future Information Systems", accepted for *IEEE Computer*, March 1991.

[Wilkinson 90] Kevin Wilkinson and Marie-Anne Neimat. "Maintaining Consistency of Client-Cached Data", *Proceedings of the Sixteenth Conference on Very Large Databases*, pages 122-133, August 1990.

[Winslett 89] Marianne Winslett, David Knapp, Keith Hall, and Gio Wiederhold. "Use of Change Coordination in an Information-Rich Design Environment", *Proceedings of the Twenty-sixth ACM/IEEE Design Automation Conference*, June 1989.