Object Oriented database subsystem for CAD frameworks

Christian Marcotte

Department of Electrical Engineering

VLSI Laboratory

McGill University, Montreal

November 1993

A Thesis submitted to the Faculty of Graduate Studies and Resaearch in partial fulfillment of the requirements for the degree of Master of Engineering.

SOMMAIRE

Cette thèse décrit la recherche et la conception d'un nouvel environnement de développement CAO ainsi que l'implantation d'un prototype.

Elle effectue une étude de la fonctionalité des systèmes existants ainsi que des besoins des environnements de CAO modernes et définit une architecture.

L'architecture utilise la méthodologie orientée object et se base sur le language "C++" dans un environnement de station de travail UNIX.

Cette thèse décrit la base d'un serveur orienté object ainsi qu'une librairie de classes pouvant être utilisée par une application pour obtenir les services suivants: persistence d'object, accès invisible et simultané à plusieurs serveurs, contrôle de versions, facilité de description d'objects complexes et possibilité de fonctionner dans un réseau hétérogène de stations de travail.

Cet ouvrage discute aussi de méthodes d'implantation de services complexes tels la notification et l'administration de project et de version à haut niveau.

Finalement, le protype est évalué et les résultats montrent qu'avec quelques améliorations de performance, ce système pourrait être utilisé à la base d'un environnement de développement polyvalent et extensible.

ABSTRACT

This thesis describes the research and design of a new CAD framework and implements a working prototype of the system.

The strengths and weaknesses of some existing systems and the needs of modern tool development are described. An architecture is defined.

The architecture uses object oriented programming methods and is implemented using the "C++" language on workstations running the UNIX operating system.

This thesis describes the basis of an object oriented server and a base class library to be used by the client applications to obtain object persistence, transparent access to multiple servers, version control, rich modeling capabilities and operations in a heterogeneous network of workstations.

There is also a discussion of methods for implementing advanced features like notification and project and version management.

Finally, the working prototype is evaluated and the results show that with some performance improvements, the system could be used as a foundation for a highly versatile and expandable CAD framework.

Acknowledgements

I would like to thank my thesis supervisor, professor Michel Dagenais for his exceptional technical feedbacks and precious advises.

I would also like to thank my colleagues Nathalie Farjallah, and Djamal Bouarab for the positive exchange of ideas they provided.

Finally, I would like to thank my friends at Bell-Northern Research for reading preliminary releases of my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis describes the foundations for a new Computer Aided Design (CAD) framework and the implementation of a working prototype of the system.

A CAD database should offer rich modeling capabilities, version control, client/server multi-user networked access, notification services and ease of integration and development. None of the systems examined perform all these tasks adequately.

The new framework is implemented in C++ [22] [46] [64] (an object oriented superset of "C" [33]) under UNIX [2] [27] [47] in a TCP/IP [14] networked environment.

Ideally the framework should provide the following services:

- object persistence mechanism,
- support for complex objects,
- version control,
- object sharing at the application level,
- concurrency control,
- distributed client/server architecture,

- inter process notification,
- ease of development for new tools,
- ease of integration of existing tools,
- automatic version translation,
- journaling,
- error recovery,
- indexing,
- query language,
- audit trail,
- rollback / crash recovery,
- portability.

All of these issues are addressed at least partially in this thesis and a thesis written by Nathalie Farjallah [23], with the exceptions of indexing and query language facilities which are projects by themselves and fall outside the scope of these thesis.

The proposed framework is divided in three parts: an Object Oriented DataBase (OODB) server, a client base class library and a network interface library for both client and server. The OODB server was developed by Nathalie Farjallah in the context of her thesis while the client and network libraries were developed as part of this thesis.

With the ever increasing complexity of Very Large Scale Integration (VLSI) designs, CAD frameworks are becoming larger every year and have encouraged various separation of functionality. The following schema is felt to be a good layering of resources for CAD frameworks as shown in Fig 1.1.

Each layer contains the following functionality:

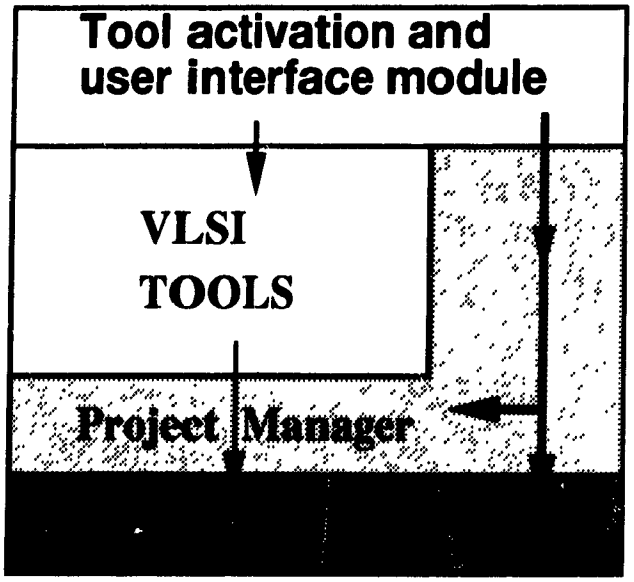


Figure 1.1: Suggested layering of a CAD framework

1. CAD OODB SUBSYSTEM: basically handles all the goals mentioned in the above list but with only an elementary version control mechanism.

2. PROJECT MANAGER: Provide more elaborate version control and concurrency access like:

   - access concurrency
   - two phase versioning
   - long time locking
   - group access
   - workspace
   - configuration
   - workspace and configuration browser
   - function library to set/modify defaults and policies.

3. VLSI TOOLS: compactor, PLA generator, router, VHDL compiler, DRC.

4. TOOLS ACTIVATION AND USER INTERFACE.

The division of the framework in this multi-layer diagram is very general and keeps any VLSI specifics to the higher layers of the model. The OODB subsystem is slightly oriented towards general CAD usage and as is the Project Manager but both are also well suited for software development projects. Specific design orientations begin with the setting of rules in the project manager. The TOOLS and Framework are definitely VLSI oriented. By keeping the lower layers as general as possible, future development is not limited by arbitrary file formats or object types.

This thesis concentrates on the first level: CAD OODB SUBSYSTEM. It integrates most of the services in the list above. The areas of special interest are tool integration and development, notification between tools and versatility in client/server distribution, the weakest areas of existing prototypes.

The prototype provides a proof of concept and the results show that while some performance improvements need to be made to the prototype, the system as it stands could be used as the foundation for an advanced CAD framework.

The remainder of the thesis is structured as follow:

The second chapter describes the following design goals of modern CAD systems and how well some existing VLSI CAD frameworks meet these goals:

- Object Oriented Database,
- tool integration and development,
- notification,
- version control and concurrency,
- project management, user interface and tool control

An emerging standard named CORBA that provides tools inter-operability is then described.

The third chapter reviews the concepts used in this project. It starts with a review of the Object Oriented (OO) concept and more specifically the C++ language. Transparent object persistence through the use of smart pointers is studied along with related subjects such as the loaded object dictionary. Since persistent objects are names using a UNIX like pathname scheme, the natural way to store a loaded object dictionary is in the form of a directory tree. This UNIX similarity is exploited to implement the mount command that provides transparent access to multiple servers. This approach is very similar to NFS mounting multiple servers to form a single directory tree. Finally this chapter will cover the topics of versioning, dynamic type checking, storage of complex objects, networking and notification.

The fourth chapter describes the implementation details of each of the features discussed in chapter three.

The fifth chapter provides an evaluation of the prototype. It describes the details of the prototype then proceeds to evaluate the memory overhead, disk usage, performance of swizzled[1] smart pointers versus regular pointers, loading/saving times of objects using the persistent object base class library versus direct NFS read/writes. Also a comparison of the original design goals with the functionality provided by the prototype is given. The chapter concludes with a discussion of interesting areas for future developments.

The last chapter provides the conclusions for this project.

[1] A swizzled smart pointer means that the object is loaded in memory and the smart pointer knows the address of that object

Chapter 2

Background

In the last few years, Computer Aided Design (CAD) systems in general and especially those targeted for VLSI, have rapidly increased in complexity. Some of the earlier projects tackled the problem of designing a CAD system as a single task. The resulting product is a single huge program like Electric [51] [52] or a set of tightly coupled programs like Oct/VEM [54] [55]. Both projects expand on some interesting ideas but leave room for work on more distributed and flexible schemas.

Due to the increasing complexity of VLSI systems, stand-alone systems can no longer do all the work. Software developed by different organizations sharing a common database repository is often required. Thus, an important goal for CAD databases has become flexibility and ease of integration [12] [30] [41].

This chapter will describe some aspects of CAD frameworks and why they should be design goals for modern CAD systems. It also reviews how well each one of these goals is fulfilled by existing systems. These goals are:

- Object Oriented Database,
- tool integration and development,
- notification,

- version and concurrency control
- project management, user interface and tool control.

Finally, there is a review of an emerging standard called CORBA which defines many services desirable in any type of framework.

2.1 Object Oriented Database

One way to achieve flexibility is to avoid specificity. The base of any framework system is a rich database. If the database itself is restricted to a small set of primitive objects that it can store, incorporating new tools with their specific formats might be difficult if not impossible.

Since the very beginning of the relational model, CAD researchers have recognized its limitations and have often used hierarchical databases [34] [61].

With Object Oriented languages came Object Oriented Databases (OODB) [34]. CAD researchers, among others, adopted this new concept rapidly because it matches their needs well [28] [10]. Indeed, OODB have the intrinsic ability to store complex objects. OODB research is a very active field with important projects like ORION [35], O2 [19], IRIS [72] and POSTGRES [62] [59] [63]. The latter implements OO through extensions to the relational model.

A basic OODB canvas includes the core object-oriented concepts reviewed in the next chapter. Like any other database, it must provide persistence for the objects and their descriptors (schema), an interface for schema definition and modification (data definition language), and a mechanism for creation and deletion of objects.

This project uses a minimalist approach to object-oriented databases which can be augmented with many additional features. Good examples of extensions to the basic object-oriented database canvas can be found in the ORION system (more specifically the ORION-1SX system)[35].

2.1.1 The ORION Object Oriented Database System

The ORION system is intended for applications that use object oriented concepts like AI, CAD, engineering and office information systems. ORION provides persistence and sharing for objects. It has a set of database features, including queries and automatic query optimization, transaction management, dynamic evolution of the database schema, multi-media information management framework, version change notification, composite objects and text search capabilities.

ORION is based on the LISP programming language [57]. To provide an object-oriented database interface, the ORION system extends the LISP language with database-related constructs. This approach simplifies the programmer's task since he only has to learn the new constructs. In relational systems "embedded SQL" is generally used to interface with programs. Such an approach requires the programmer to learn two languages: the host language and SQL. Furthermore, the programmer is responsible to adapt the data model of SQL and the data structures of the host language.

ORION is a networked database that runs on local area networks of UNIX workstations (Sun and Symbolics). Its client/server configuration adopts a star configuration as seen in Fig.2.1. The server provides concurrency control and locking on the objects. Communications between clients and server are done via Remote Procedure Calls (RPC) [15]. The server uses a storage manager that does not use UNIX file system but rather manages its own raw disk partitions, providing improved access efficiency.

It supports versions for objects and distinguishes amongst *transient versions* (temporary versions), *working versions* (stable versions) and *released versions* (stable and debugged).

It has a change notification mechanism for references. This type of notification is suitable to warn users of modifications in an object referenced by one of their own objects. It is not suitable for tool notification, which is necessary when many tools work concurrently on the same object.

The ORION system uses only a simple write ahead log scheme that allows it to recover from a soft crash (which leaves the content of the disk intact). It does not however support archival dumping or mirroring of the database to allow recovery from a hard crash (which

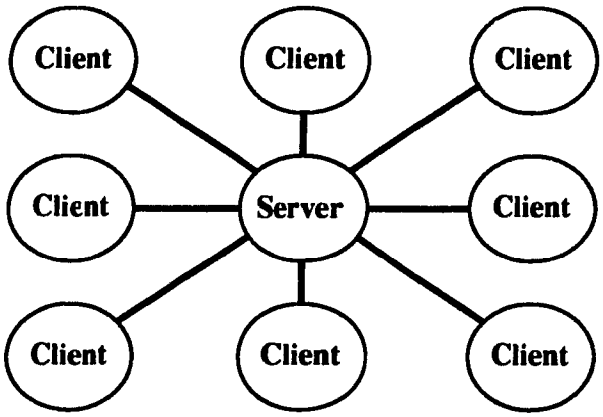


Figure 2.1: ORION client/server configuration

destroys the content of a disk).

ORION being a research prototype, with limitations on manpower, did not address support for multiple application programming languages, such as "C". This is a major shortcoming in the CAD environment where most programs are written with the "C" language.

2.2 Tool integration and development

Previously, tool integration was done using filters that converted between incompatible formats. One way to minimize the number of filters needed by the framework is to provide one filter to a standard file type like the "Electronic Design Interchange Format" (EDIF) [39]. File translation and exchange is the most primitive system. It is limited to tools that work in batch mode and makes it difficult to implement error reporting between the communicating tools.

A more efficient integration requires the new tools to communicate with the CAD framework database. In such a case, the ease of integration varies with the framework depending on its internal organization. Most frameworks require at least partial rewriting of the tool to integrate.

At one end of the spectrum are the CAD frameworks like Electric [51], which require integration of the new tool in the executable module of Electric itself. This requires considerable rewriting of the tool and extensive knowledge of the internals of the host framework along with its database. Such an approach is not practical in today's CAD environment because of the large number of tools that are needed to achieve a complete CAD framework. It is common for a VLSI CAD framework to offer many tools with similar functionality, like various flavors of place and route tools. Typically only one of these tools needs to be used during a work session. Since every tool needs to be incorporated into a single executable, the memory usage would greatly suffer.

A compromise approach is taken by other systems like Oct [26] that use a custom implementation of the Remote Procedure Call (RPC) rather than direct linking to the framework's core executable. Oct provides interface functions [56] to the database system and the framework graphics system. This allows the new tools to run in a different address space than the server (Oct core). This way, tools run as different processes either on the same machine or another one accessible through the network. A major advantage over Electric's core integration approach is that the framework does not have to be recompiled to integrate a new tool. However, Oct still restricts its representable objects to a limited set of VLSI oriented primitives, and all object manipulations are done through a predefined set of functions. The user has to recode its application to use Oct's object model and object manipulation library. In other words, this system does not provide storage and primitives for the user's objects, but rather expects the user to convert its object to an Oct format and manipulate them through a well defined Oct interface.

Others more moderate frameworks require adding the appropriate database access function calls in the tools to integrate, fill schema definition files to feed some special preprocessor or other similar schemes [30] [12]. Such an approach allows the user to create and manipulate its own objects, but still requires the programmer to understand the details of the database mechanism, and adapt its data structures to the data model available. Some systems using this scheme have reported on an average of four man-months of integration effort per tool [12].

An alternative to a "linked" interface to the database is used by certain systems like

Cadence. Cadence provides interfaces to its database through a proprietary LISP-like interpreted language named IL/SKILL [11]. This interpreted language allows the framework user to build complete programs using the IL/SKILL language. In this language however, the only possible I/O with the outside of the framework environment is via UNIX files. In other words, this is a wonderful integration mechanism, but it only integrates the tools on a file exchange basis. Such loose coupling integration does not satisfy the need for inter-tool communication and fast database access. Named pipes appear like files to the applications using them, hence can be used to speed up Cadence's tool interfacing.

All the above methods require considerable effort and knowledge of both the new tool and the framework from the programmer. They also limit integration to the fixed schema supported by the database. An object oriented system like ORION allows easier integration but requires that the tool to integrate also be written in a compatible object oriented language. Table 2.1 summarizes the different tool integration approaches discussed above. Whatever method or system is used, interfacing a new tool to a framework is rarely a trivial task.

As important as third party tool integration is tool development. Without automated support for persistent objects, the tool developer is responsible for translating the internal data model into some suitable disk representation and using or implementing the data access functions. This amounts to a considerable amount of effort on the part of the framework user [28]. This is why the elementary file exchange methods that is the simplest form of tool integration requires more work for tool development. Many systems (Electric, Oct) that provide a somewhat rigid object model and related access functions greatly speed tool development by cutting data representation and storage development overhead. However, both limit the tool to the object manipulation and representation facilities built in their respective systems. The object oriented approach is the simplest and most versatile since it allows the programmer to use his own objects and internal object manipulation methods by providing persistence with language extensions. Object persistence is described in section 3.3.

Type	Description	Disadvantages	Advantages
file exchange	Writing various filters between different incompatible file formats. May require recompiling of the framework	• slow • batch tools only • multitude of filters needed	• easy to do
file exchange with EDIF	Writing filters to and from a common file format (EDIF)	• slow • batch tools only	• easy to do • no recompiling of framework
file exchange with interpreted database access language (Cadence)	Interpreted language allows versatile filter building without recompiling tool or framework.	• slow • batch tools only	• fast integration in high level lang. • no recompilation of tool or framework
core integration (Electric)	Recompile both the framework and the new tool in a single executable file.	• long integration • requires high expertise in tool and framework • large portion of tool needs recoding • generates very large executable	• highly efficient • allows easy tool notification
RPC library (Oct)	Keep object in local memory but allow remote manipulation via RPC.	• requires expertise in framework • large portion of tool needs recoding	• no recompiling of framework • more efficient than file exchange
various preprocessor methods + function integration	Varies from one system to the other	• non transparent scheme • requires adapting tool schema to framework data model	• tool keeps using its own object format internally
object oriented language extentions (Orion)	Makes language extentions to (almost) transparently provide persistency.	• tool has to be written in an OO language	• almost transparent persistency, hence very easy integration • fast database access

Table 2.1: Tool integration methods

2.3 Notification

In an integrated CAD environment, many tools have to cooperate and work on the same data. Moreover, some of the tools are incremental (i.e. they do their work based on changes issued by other tools). An example of this is an incremental critical path evaluator [4] used in some VLSI CAD frameworks. The designer modifies a cell through the graphics editor which notifies the critical path evaluator when a change is made. The critical path evaluator then reevaluates the critical path.

Some frameworks have data notification but it is limited to their own set of internal tools running on a single host like Cadence [11] and Electric [51]. Oct [26] [53] offers something it calls notification, to external tools running as different processes (possibly on a different host).

The next two subsections describe the Electric and Oct systems with emphasis on their notification capabilities.

2.3.1 Notification in Electric

Electric is a single user non networked VLSI CAD framework. It is written using the "C" programming language in a UNIX workstation environment. Current versions run on the X windows system. It uses a common database for all the tools integrated in its framework. The database contains objects related to VLSI CAD that form libraries. Each library of objects is represented by a UNIX file.

In the Electric framework, all available tools are compiled in a single executable program. When loading objects from disk, Electric loads the whole file (library or design project) in memory in a single operation resulting in a long loading period. Both these approaches of having a large executable with all tools integrated and loading the whole design/library in memory require large amounts of resources from the host system.

Figure 2.2 illustrates the internals of Electric. The core of Electric is its database. It is accessed by a predefined set of functions that manipulate the objects in memory. The

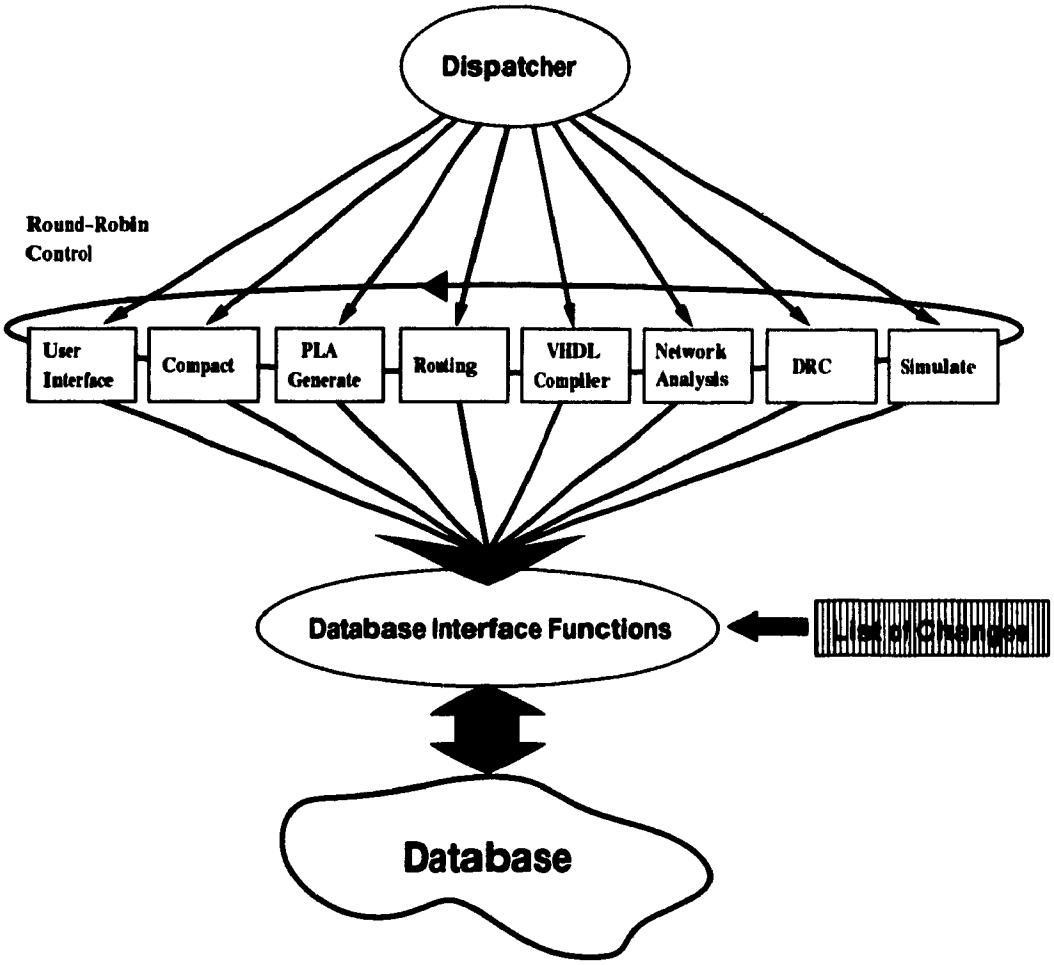


Figure 2.2: Internals of Electric

function library is somewhat simpler than the one provided by other systems since it does not have to worry about whether an object is loaded or resides on disk since all the design's objects are loaded in memory at startup time.

Tools are built as functions that call the database manipulation functions. Controlling the tools is a dispatcher which acts like a primitive non preemptive scheduler that gives running privileges to each tool in a round-robin fashion.

A tool can either be **on** or **off**, in which case it simply skips its turn. Since every modification to the database is done through a well defined set of functions, the database

keeps tracks of what has been modified and when the dispatcher gives running privileges to a tool, the latter can then be informed of what happened while it was asleep. This is called *notification.* It is made possible in a relatively painless manner since the framework is dedicated to a single design operation and all tools share the same memory representation of the design.

In a networked environment, the technical challenge of notification is greater as illustrated by the Oct Framework which is discussed next.

2.3.2 The Oct notification system

The Oct system is a VLSI CAD framework originally developed as a single designer system. It was later enhanced with the *Octane* package providing it with versioning and concurrent access control. Octane's goal is to provide for group development.

Oct is written using the "C" programming language on UNIX workstations. It has a tightly coupled graphics editor called VEM. The latter acts both as a graphics editor and user interface to control the other tools. The graphics interface uses the X windows system.

Oct is a data manager (or database) that acts as a central repository for all the design data. The objects have an unique internal identifier, and a name resolution module is used to map those identifiers to UNIX pathnames. It supports a limited set of VLSI primitives that can constitute objects. To create, modify, manipulate, load and save objects, Oct has a function library general enough to suit the needs of most VLSI tools. This library is based on a customized RPC system. It allows tools to reside on a different host and obviates the need to recompile the framework every time a tool is added.

An RPC function behaves much like a regular function but in a different address space. One difference with Electric is that Oct does not dispatch running time to its tools. The tools can be asynchronously started on one or more machines and concurrently access the central Oct server. Figure 2.3 illustrates the Oct mechanism (without the Octane package).

Even though tools can be dispersed throughout many machines, they all share a common central memory image of the objects on the Oct server. Since tools run concurrently,

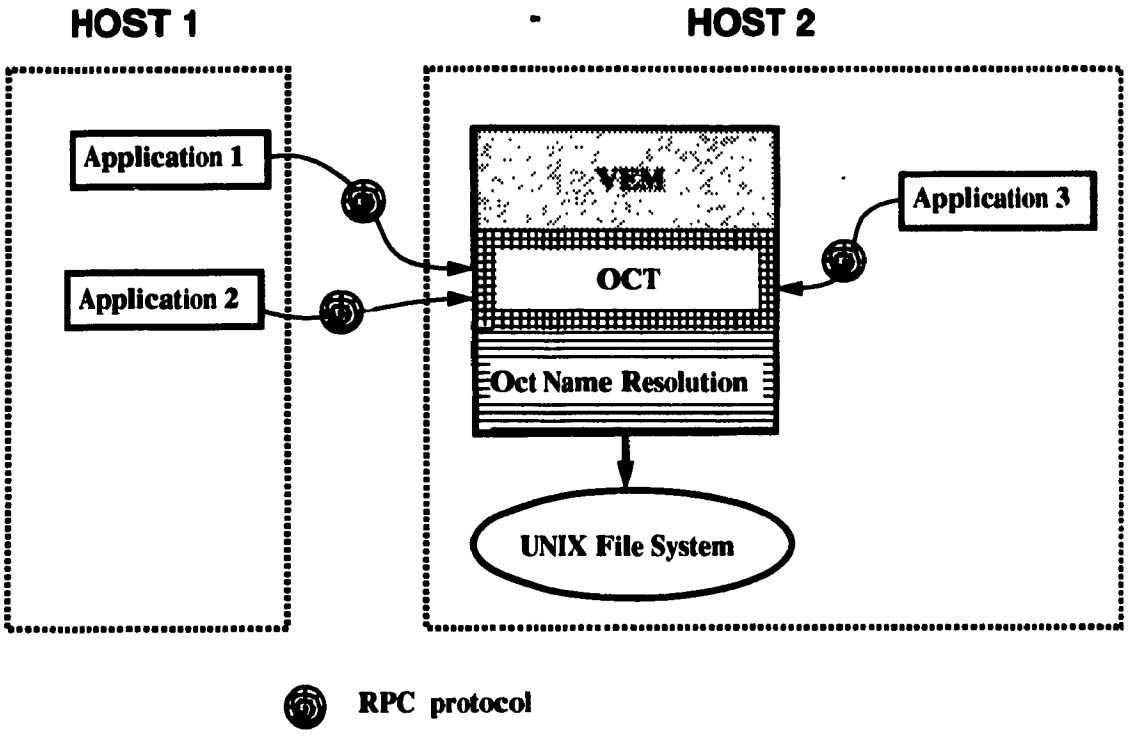


Figure 2.3: Oct mechanisms

no dispatcher can hand them a modification list upon awakening.

In the Oct system, every action performed on an object is done via a library function. Every function that modifies an object changes its timestamp and logs the action in a *change list.* Every tool interested in possible changes in the design must periodically poll either the change list or the timestamp of interesting objects.

This is not a true notification mechanism. Indeed, as implemented here, notification is associated with an interrupt mechanism rather than polling. Although a polling mechanism simplifies the implementation of the "notification" system, constant polling brings about an unnecessary load on both the server and the tools.

An important problem with keeping and manipulating all objects on the server is that in a multi-user environment, it becomes cumbersome and inefficient. With today's explosion of VLSI chip complexity, the number of objects that a single user can request may be

overwhelming.

2.4 Versions and concurrency control

As projects grow bigger, more designers work in parallel. The frameworks/database become networked client/server processes. To insure data integrity, new multi-user systems need versioning and concurrent access control.

To satisfy this need the Version Server research project [32] concentrated on versioning and concurrency control. Many research efforts were inspired by this work [6] [20] [21] [71] and notably [3]. Some of these principles are implemented in major projects like ORION [13] and Octane for Oct [53]. These principles and their implementation are discussed below.

2.4.1 Two Phase Versioning

In a CAD environment, a simple versioning mechanism is often not enough to support the needs of different groups working with the same objects. A more sophisticated two phase versioning/locking mechanism is needed [13] [6].

The first phase is a simple monotonously increasing version number and locking mechanism for elementary objects, as can be found in source code control systems like *SCCS, RCS* and *CVS* [50] [67] [5].

The second phase has to do with who wants to use the objects. This second phase has been divided in three stages:

1. transient (for developer of the object)
2. working (for technical users of the object)
3. released (for public Release)

The following example will illustrate the need for those three stages. In a company that designs VLSI standard cells and libraries, many groups work in parallel. A group often uses some cells under development by another group as building blocks in their own project.

The group making the basic cells modifies them on a daily if not hourly basis and cannot guarantee the functionality of the cells at all times. This group works on a **transient version** of the cell, the most up to date but still bug ridden version.

The second group needs a cell version that is relatively stable, and guaranteed to be functional, yet recent enough to have all the new features. They need a **working version** of the cells for their project. Naturally, a few bugs might filter in the working version but the groups communicate bug reports, and corrections are quickly made.

The third user group is the customer. The customer should not have access to anything that has not been thoroughly tested. The customer will have a **released version** of the cells. Table 2.2 illustrates the relation between the users and the two phases of versioning.

It can be noted from the figure that all version numbers get to be transient versions. Since not all versions are fully functional, **working version** skips a few version numbers between each "promotion". Since bugs still exist in some working versions and a released version has to be more stable, the latter skips a few working versions between each promotion. In other words, during the life of a design, there are many transient versions, fewer working versions and still fewer released versions.

The simplest way of providing this notion of current version oriented toward the user is to have both design teams work on different servers and make physical copies of the version as it is ready to be promoted to the higher level. This method is error prone and has no support for history of the second phase version progression.

An intermediate approach is to use a *check-in* mechanism on a "group server" that contains the working copies of the objects. The transient versions are kept in a local database used strictly by the designer of this set of objects. This allows an effective distribution of the resources over designer and group server but still lacks a complete history mechanism of all version levels on a single server.

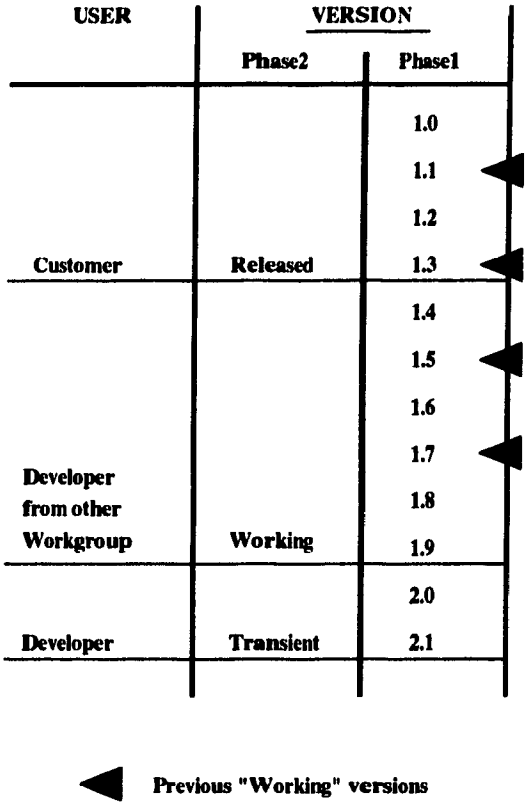


Table 2.2: Users and two phase versioning system

A more organized method is to use a single server. Such a system requires a two phase version/locking system. It can be implemented by simply using a table (see Table 2.3) to keep track of the working and the released version number for a given object. The transient does not need to be in this table since it is always the highest numbered version.

2.4.2 Long time locking and group access

Typical CAD engineering designs will take many days if not many months to complete and involve many designers. Locking of objects for the duration of the project while granting access to a selected group of designers is a much needed feature for a CAD design system.

Members of the design team are allowed access to the design during the extended lock period but a lower level of locking must still be in effect to prevent two designers from

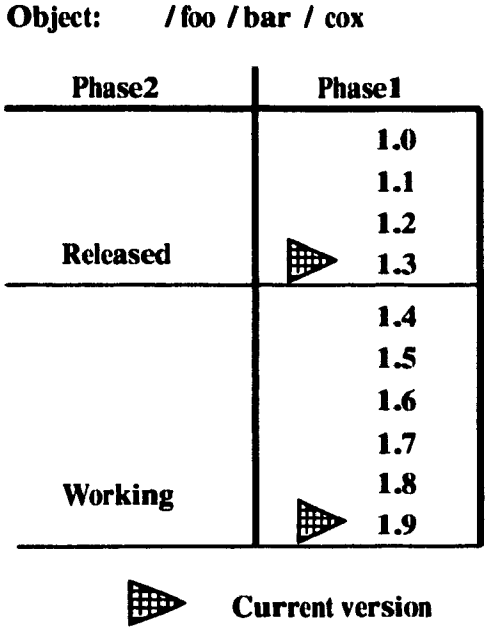


Table 2.3: Table implementation of two phase versioning system

modifying the same object simultaneously. An elementary object locking for a work session period can easily be provided by the first phase of a two-phase versioning mechanism. However, a higher level access management system is needed to provide a long time locking and group access.

2.4.3 Workspace

A workspace is a grouping of many objects to be recognized as an entity. Its purpose is to allow a higher granularity of locking. Hence locking a workspace with certain privileges is the equivalent of locking every individual object of this workspace.

A workspace can easily be implemented over elementary versioning/locking systems with a book keeping layer that should be inserted between the client and the database.

2.4.4 Configuration

A project configuration is like taking a snapshot of the design at a certain point in time. It can be viewed as an internal milestone or a product release version. It may also represent a specific release targeted for a given environment, for instance a workstation or a personal computer configuration. It is a higher level of versioning for a whole project.

A configuration merely takes note of all the objects that compose the design and their respective version [53] [32].

2.4.5 Version management in Oct using Octane

Octane is the versioning mechanism for Oct. It implements most of the principles presented above. Figure 2.4 illustrates where Octane fits in the Oct library. Octane brings protection to a group of objects hence bringing the locking granularity to a higher level. It also performs version resolution and understands the notion of configuration. Octane was implemented as an extension to the Oct library. The Oct library functions that formerly accessed the Oct name resolution module now access the Octane name resolution module.

The Octane name resolution module sits on top of the Oct name resolution module. The latter only copes with "direct" orders in the sense that it matches an object identifier with a UNIX pathname. It has no notion of policies or default values, it simply makes a direct match between two identifiers.

The Octane name resolution module acts as the policy oriented module. When an application requests an object without specifying a version number, Octane will select one according to its internal policies. Since Octane filters all calls to the Oct name resolution module, it can add functionality for workgroups and configurations. If a workgroup or configuration is specified, Octane can expand its name to each of its member's names and versions number before passing them to the Oct name resolution module.

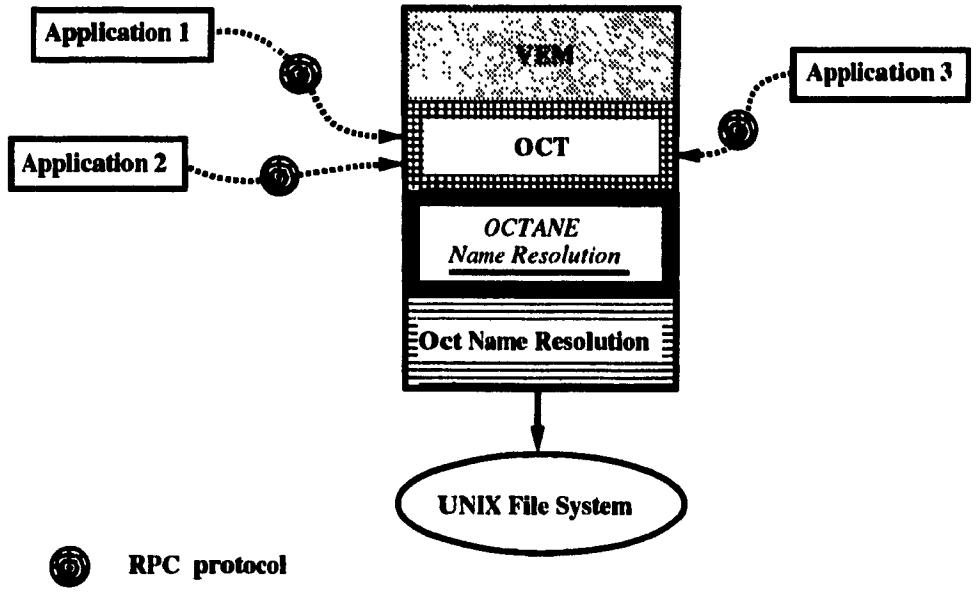


Figure 2.4: Octane integrated to Oct

2.5 Project manager, user interface and tool control

Efficient project management and design browsing facilities are required [41] for larger projects since, more data, more tools and more people are involved.

2.5.1 Project Manager

The Octane package that was briefly described in the previous section, implements such data management (or project management) policies. Policies form a set of default values and strict rules to be enforced on tools accessing the database.

Since design policies vary from project to project and can even be modified in the middle of a project, hard coding a set of policies in any given project management tool seems like a bad idea. With this in mind, the Octane library was built with a versatile rule (or default) setting interface. The previous section only shows Oct's name resolution module, assuming that it uses hard coded rules. Figure 2.5 (figure from [53]) illustrates the full Octane interface in the Oct context. Normal tools interface with the Oct library,

which in turn will pass its requests through the Octane name resolution layer.

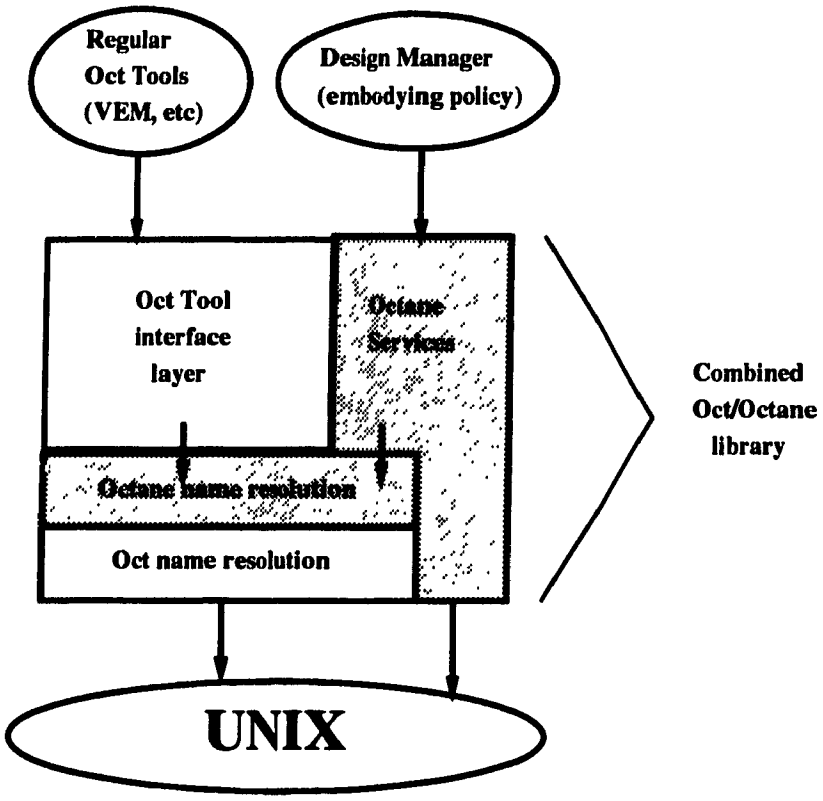


Figure 2.5: Octane's Design Management interface

The Octane library provides some functions to set the defaults and rules called *Meta Design Data* used by its name resolution module. It also provides high level functions to set, revise and browse through workspaces and configurations. These functions can be accessed at the application level just like Oct functions. The Oct functions are accessed by the design tools that know nothing about policies or other high level information. The Octane functions are accessed by Design management tools. These are used by the project leader to set and revise policies and by designers to browse through workspaces and configurations. This approach cleanly separates the design tools from the design management tools.

Other high level Design Management tools from other research projects can be interfaced to the Octane library, like a graphical design browser [25] or a change coordination

system [73].

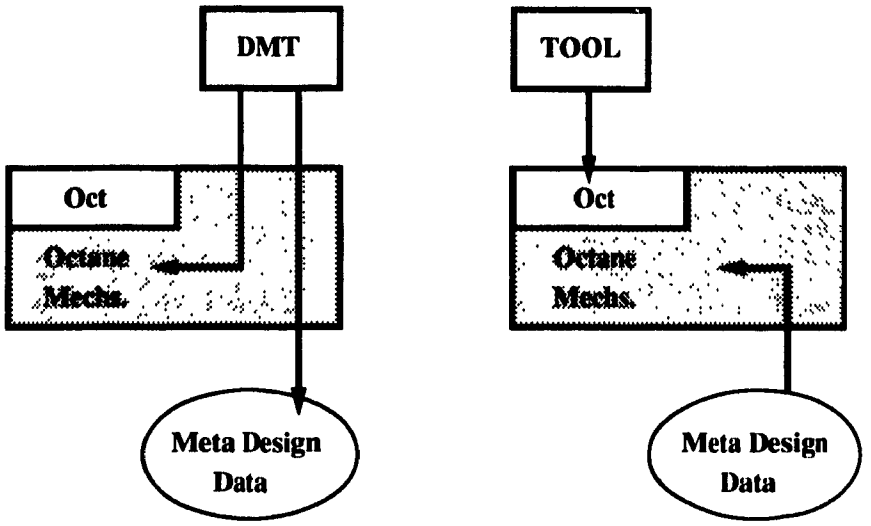


Figure 2.6: Octane's Meta Design

Figure 2.6 (figure from [53]) shows how the Meta Design Data is set by the Design Management Tools, stored on disk and later loaded for future design sessions.

2.5.2 User interface and tool control

With many tools to control, the user needs a uniform user interface. Work on user interfaces to control local and remote tools in a uniform way have generated tools like VEM for Oct [26] and DEBBIE [74] for HILDA [30].

Other interesting work on general framework integration include [7] [30] [69].

2.6 CORBA

While CORBA is not specifically a CAD framework, it could be part of one. In 1989, a group of hardware and software vendors created the Object Management Group (OMG)

to address the problem of object interoperability in distributed applications. In 1991, the OMG came out with a reference model and the Common Object Request Broker Architecture (CORBA) which describes the interfaces and the services provided by the system [45].

2.6.1 An Overview

At the heart of CORBA is the Object Request Broker (ORB) which provides the interface between the client and server objects. CORBA provides an object messaging system that is language independent, location independent and platform independent.

To interface with a CORBA compliant ORB, a client must use an Interface Definition Language (IDL) which is then compiled in the language of choice to merge with the user's application. CORBA is independent of the machine architecture or the operating system on which the objects run. This means that an object coded in "C" on a UNIX machine can access a service provided by an object coded in FORTRAN on a VMS host on the network. The location independence is also provided by the ORB. Given a request for an object, the ORB is responsible to find it. The object can be on the same machine or anywhere on the network. An ORB can even be interfaced with an X.500 service to locate objects. Once located, the ORB is in charge of passing the request to the server objects and returning the results to the client.

The ORB uses a registry of servers called the implementation repository (also called interface repository). In order to be used, it must be registered with the repository. Once registered the server can be re-launched if it is not already running. This registry can also be queried to get a list of all operations supported by an object.

In addition to the IDL, a client can access services through the Dynamic Invocation Interface (DII). The IDL and DII can be compared to queries to a relational database using either embedded SQL or interpreted SQL. The former is hard coded in the program but very fast while the latter can allow a user to interactively formulate creative queries, but is much slower due to the dynamic parsing and type checking overhead. The DII combined with the ability to list services provided by different servers can allow a client

to use services that did not yet exist when it was written. Such services could be a new editor or debugger that do not require critical communication performances.

There exists a layer called Object Adapter (OA) between the ORB and the server object. The Object Adapter can either interface with an object written for CORBA which has an IDL interface or with legacy systems which were written before the existence of CORBA [31]. This latter kind of interface can vary widely. The OA supports a range of interfaces from modern objects with multiple member functions to non object oriented services where each function can be a separate program or a shell script.

The implementations of systems compliant to CORBA are called Distributed Object Management Systems (DOMS). Many vendors are coming out with such systems including HP's "Distributed Object Management Facility" (DOMF) (lower layer of SoftBench TM) and Sun's project "Distributed Object Everywhere" (DOE).

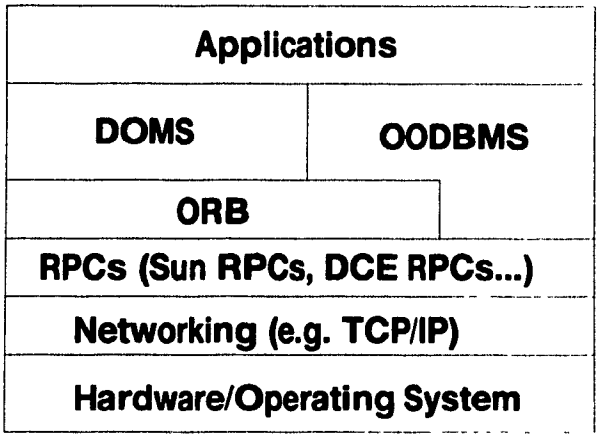


Figure 2.7: Relationship between the communication technologies

Figure 2.7 illustrates where the ORB sits with regard to other object technologies like the OODB and the other distributed computed schemes. The services provided by the DOMS are at the same layer as the OODBMS. While the DOMS currently provides a good solution to encapsulation of legacy systems and the OODBMS provides good efficient data access, the difference between the two will become less distinct as DOMS will start providing services like backup and restore, change management, concurrency control and OODBMS will start incorporating CORBA communication mechanisms.

Because the DOMS sits on top of the communication layers (RPCs, TCP/IP, OS...) and it uses an IDL compiler to interface with the application, it is expected to make applications more portable and to shield programmers from the network programming burden.

2.6.2 Services

The first release of CORBA intentionally left some areas undefined. One of those areas is a standard set of object services. The OMG is is the process of agreeing on a common set of object services [43] [48]. Those services are:

- naming (to help developers to find objects on the network),
- event notification,
- lifecycle (facility to create, destroy, move and copy objects),
- persistence (automatic object persistence).

This common set of object services is expected to grow in the future and possibly include:

- security,
- relationships,
- transactions,
- concurrency control,
- externalization,
- data interchange,
- licensing,
- trading,

- query.

Common facilities like browsers, print spooling, electronic mail and help facilities are also expected to be available through CORBA objects.

2.6.3 The current state of CORBA and its future in CAD frameworks

The current version of CORBA (v1.1) defines interoperability of heterogeneous software components within the context of a single ORB vendor, and portability of source code across ORB vendors. Also, it only specifies mapping between IDL and "C". Future versions will address interoperability between ORBs of different vendors and mapping to other languages (C++ being the highest priority).

Integrating of existing applications as clients of CORBA compliant systems, the process requires rewriting large amount of code and restructuring in order to accommodate the IDL interface.

The performance issues have not yet been observed (there are still very few CORBA products out on the market) but it can be expected that the first implementations will be slow and unoptimized. Even after a couple of iterations, the CORBA systems will be slower than today's "hard wired" systems because of the added layers and added level of indirection (everything goes through the ORB).

However, the promise of standard interoperability between objects will certainly outweigh the performance loss and CORBA is likely to be part of future CAD frameworks. Many of these concepts addressed by CORBA are described in this thesis.

More information on CORBA is available in references [29] [70] and [45].

Chapter 3

The Concepts

This chapter will introduce the concepts developed in this project. It begins with an architecture overview of the project as a whole and continues with a more detailed study of the underlying concepts. These concepts include:

- Object Oriented and C++,
- persistence,
- loaded objects dictionaries,
- UNIX-like mount mechanism,
- the object database server,
- versioning,
- dynamic type checking,
- storage of complex objects,
- networking and
- notification

3.1 Architecture overview

Some of the driving goals of this project are ease of tool integration, client/server architecture and support for data notification between networked tools.

The major tool integration methods were reviewed in the previous chapter and summarized in Table 2.1. The most powerful and natural tool integration method is the creation of extensions to an object oriented language. Its main drawback is the necessity to code the application in an object oriented language compatible with the extensions.

Most CAD programmers are familiar with the "C" programming language. Indeed, most CAD applications are written in "C". Since C++ is an extension to "C", any "C" program is also a C++ program. Thus, a "C" program can easily use C++ extensions to provide database access and persistence. With this in mind, it was decided to use this method for tool integration and development. This approach is by far the most transparent one and requires minimal intervention from the programmer.

Few CAD frameworks provide adequate notification services and fewer allow multiple servers:

- Electric is not networked and uses a core integration method,
- Oct has clients using remote procedures but keeps all objects in the server's memory with only remote manipulation allowed,
- Cadence EDGE acts like Electric since all notified tools are core integrated. The only networked support is for the library server and batch tools.

All have adopted a centralized object repository in the server, making notification much easier since all tools share the same memory image of the objects. They do not allow external tools to benefit from notification services nor permit the creation of new objects types.

In the proposed object server, the objects are encapsulated. The server's knowledge of the object is restricted to it's name, version and timestamp. Notions of ownership and other policy oriented mechanism could be added as an interface module.

This architecture is based on a minimal object server providing basic mechanisms while being ignorant of what the object is. The client (the tool) is not tied to a single server and can even access many servers simultaneously. A client registers with the server that stores the desired object and gets a copy of that object under read-write or read-only access permission.

Since each application gets its own copy of the object it must cooperate and notify the server when an object is modified. Clients can register to receive change notification on a given object when reading it from the server. Some other client modifies the object locally. When the modifications are satisfactory, it notifies the server (sends it an updated copy of the object) which in turn notifies the registered clients.

3.2 The OO concept and C++

This section will give the reader the basic concepts needed to understand the ideas in this thesis. First, elements of Object Oriented languages terminology and some peculiarities of C++ are examined, on which lie some aspects of this project.

3.2.1 The OO concept reviewed

In an Object Oriented language all conceptual entities are modeled as *classes*. A class contains attributes, much like a struct in "C" or a record in Pascal. This is what holds the information. A class also has *methods*, i.e. *functions*, that allow the class to interact with its environment. All attributes of a class are available to the class' method as if they were local variables. Having both attributes and methods together in one entity is called *encapsulation*.

The designer of the class decides what attributes and methods are visible outside the class, the others being kept for internal use. This feature is called *data hiding*. The attributes not visible outside their class are often indirectly accessed through a method that offers a cleaner interface. It is then possible to later change the attributes of the class as long as the interface remains the same.

A class is a new type just like integers, characters and floating points. The instantiation of a class is called an *object* (or an instance of the class...) much like an instance of int is a variable.

When many classes share a certain amount of common attributes and methods, these common attributes/methods are grouped into a *base class* from which more specialized classes are *derived.* The derived classes *inherit* (share) the methods and attributes of the base class.

An example of this would be the base class "car" from which one could derive the classes "convertible" and "four_door". Both *derived classes* (car model) share all the features of the base class (basic model) plus the added options. This feature is called *inheritance.* Deriving a class from another creates a *class hierarchy.*

Another important notion of OO languages is *polymorphism.* It gives an action a name that is shared through a class hierarchy, with each class in the hierarchy implementing the action in its own appropriate way.

For example, the base class "circle" has the attributes "position, diameter" and a method "display" which displays a circle. From this base class a new class called "ball" is derived. It contains the attribute "color" and the "display" method shows a colored circle. Both class circle and class ball have a method "display" which does the same action in both cases but tailored to the needs of each class.

For more detailed information on the object oriented concepts the reader should read one of the many good books on Object Oriented Design [8].

3.2.2 The context of the C++ language

One of the important features of C++ is the notion of *constructor/destructor.* A constructor is a method called automatically when the object is created (a new instance of a class). Constructors can guarantee some kind of default initialization when needed. The destructor is a method called automatically just before an object is deleted. The destructor can serve several purposes, like tying loose ends in a linked list when one of the elements

gets removed.

Another important feature of C++ is *overloading.*

> Overloading allows multiple functions with the same name to be defined provided their argument lists differ sufficiently for calls to be resolved. By overloading operators, the programmer can redefine the meaning of most C++ operators [22].

This means that the meaning of certain operators can be redefined when they act upon a certain class. This also means that many functions may exist with the same name. The compiler will decide which one to call by looking at the types in the argument list. Of course, all functions that share the same name should have the same semantics.

A class can be a data member (an attribute) of another class as it is true that a "C" struct can be a field of another field. This has implications for constructors, destructors and structure (class) assignments (copy).

When performing an assignment to an initialized object, the right hand side of the assignment operator "=" must be recognized by an overloaded version of the assignment operator. If none exist, the system will look for a constructor that would accept the argument. If one is found, the destructor of the object is called and then the object is reconstructed by the appropriate constructor.

A special constructor named *copy constructor* is automatically provided by the compiler. It is called to initialize a newly constructed object with the values of another one. Ex: For a given class Ball, Ball mine = yours; where yours is an instance of Ball. When doing an assignment like "A = B" where A and B are instances of the same class, the system does not call the copy constructor, even if there is no overload of the assignment operator. The system performs a *member wise copy* (as opposed to a *bitwise copy.* A *member wise copy* is a copy of one object to the other performed "member by member". If one of the members is a class, a "member by member" copy will be performed too. A *bitwise* copy would simply make a bit image copy of the memory space of the object. If an overload of the assignment operator is available, it will take precedence over the *member wise copy.*

Suppose a class B derived from base class A, a pointer of type base class A (A* ptr1;) and an object instantiated from the derived class B (B foo;). The following statement is legal (ptr1 = &foo). In other words, a pointer to a base class can legally point to any class derived from this base class. The converse, however, is not true.

A *virtual* method of a class can be redefined in any of its derived classes and is activated according to the type of the object, not by the type of the pointer. It is with virtual functions that there can be polymorphism in C++. This means that when we refer to a virtual method of an object, the most specialized definition of that method will be called (i.e. the most derived class that has a redefinition of the function).

Let's look again at the previous example with base class A and derived class B, but this time with a pointer to the derived class (B* ptr2;) assigned to address of an object (ptr2 = &foo;). All methods defined in base class A would be available to use as ptr2->*function-name*, in addition to the methods defined in derived class B. On the other hand, the first pointer ptr1->*function-name* could only access the functions defined in base class A. This is because it does not know about derived class B. One way to make functions from derived class B available to a pointer of the base class is to define them as virtual in the base class. Then, ptr1 can call the methods redefined in derived class B.

This notion is very important as many classes are derived from a base class. Since the base class is the only common ground between all the derived classes, pointers are declared to the base class type. Methods defined in derived classes can be accessed properly when defined as virtual.

Finally, the C++ language has recently acquired a new facility called *templates*. One type of the templates of interest here, is the *parameterized classes* or class templates. A class template specifies how individual classes can be constructed much as a class declaration specifies how individual objects can be constructed.

Traditionally if one writes a "linked list" of objects of class "coconut" that contains a pointer of type (coconut *) and wants to make a similar list but this time of class "pumpkin", he must copy the class definition and change the (coconut *) to a (pumpkin *), all this to satisfy type checking at compile time. One possible solution is to use opaque

pointers (void *) but it prevents meaningful type checking.

Using templates, there is an argument <T> to the "linked list" class definition. Within that class is a pointer of type (<T> *) which is used where <T> serves as a parameter for the class template. This is particularly useful for container classes like trees, lists and bags.

For a more detailed reference on C++, the reader should consult the language definition and reference manual [64] [40] [22].

3.3 Persistence

Persistence is the ability of complex memory objects to survive a program invocation [9] [18] [38] [17]. Programmers have used many ways of saving and restoring different kinds of complex objects like "C" structs, arrays, and linked lists but these mechanisms are often type specific and the programmer spends great efforts reimplementing these mechanisms in every program.

Transparent persistence support for complex objects could easily form the base of a CAD data repository. Ideally this would be done by the compiler for complete transparency. However unlike Eiffel [42] and Modula-3 [44], most mainstream languages like "C" and "C++" do not offer such support. Persistence in C++ can be achieved via some language extensions [49] or other clever mechanisms [1].

There are mainly two challenges: Resolve references to complex objects stored on disk and implement the persistence in a transparent way.

3.3.1 Pointer representation

References to complex objects are often kept as pointers, a memory address that changes from one program invocation to another. Thus, an alternative representation for references to persistent objects must be found. The access method should not change whether the object is on disk or in memory.

Many schemes are possible: using a unique integer as object identifier, called external data identifiers (XID) [55] or using symbolic names. A symbolic name, similar to the UNIX pathnames, where a name is composed of many tokens separated by "/" is used in the system proposed here. ex: /foo/bar/cox. UNIX users are familiar with this hierarchical naming convention. It allows the user to assign meaningful names to objects, as opposed to arbitrary numbers. Thus, the reference to a persistent object will use its pathname.

3.3.2 The Smart Pointer

A smart pointer contains the name (pathname) of the object it references, a state and possibly the memory address of the object, depending on the state. A smart pointer can be in one of two states: swizzled or unswizzled.

A swizzled smart pointer is one with a valid memory address pointing to the object loaded in memory. An unswizzled one has only the name of the object. Whenever someone tries to access a swizzled smart pointer, the memory address serves as an ordinary pointer. When an unswizzled smart pointer is accessed, it calls the loading method to get a valid memory address and then becomes swizzled.

Since many smart pointers may reference the same persistent object, the object can be loaded in memory without all smart pointers referencing it being swizzled. If a persistent object gets unloaded from memory, unswizzled smart pointers remain unaffected but the swizzled ones would contain a memory address that is no longer valid.

To insure the integrity of smart pointers, a persistent object must notify all swizzled smart pointers referencing it when it is being unloaded. They will then change their status to unswizzled. To do this, the persistent object must maintain a list of the swizzled smart pointers referencing it. This brings up a second level of integrity. When a swizzled smart pointer gets deleted, it must notify the persistent object to remove it from the list.

The smart pointers replace traditional pointers when referencing persistent objects. Pointers to non persistent objects remain unchanged.

The cost of using smart pointers is relatively small. In terms of storage space it needs:

1. OBJECT NAME + OBJECT ADDRESS + BACK POINTER TO SMART POINTER.

Performance wise, the cost of a smart pointer is:

1. If swizzled (most frequent case): CHECK IF SWIZZLED.
2. If unswizzled, loaded: CHECK IF SWIZZLED + SWIZZLE.
3. If unswizzled, unloaded: CHECK IF SWIZZLED + LOAD + SWIZZLE.

The storage cost is the most significant of the two and can be reduced by mapping the pathnames to a numeric value. The CPU cost is very small. The swizzle and load time are advantageously compared with traditional reading from a disk file. It takes no more time and allows loading the objects on demand as opposed to loading all possible objects at the beginning of the application. A swizzled pointer incurs very little CPU overhead, simply one comparison. The loading and swizzling are infrequent since a smart pointer usually gets loaded/swizzled only once but is referenced many times.

The overall cost is minimal, considering the flexibility gained, since objects can be loaded or unloaded from memory dynamically.

3.3.3 Transparent access to Persistence

By using C++ operator overloading facilities, smart pointers can be supported transparently. By transparent is meant that once initialized, the smart pointer behaves like a regular pointer.

The persistent objects still need special attributes: a name and the back pointer list (of swizzled smart pointers). Also needed are access methods to load, unload, save or delete the persistent objects.

In order to achieve almost transparent persistence for objects, the class derivation mechanism of C++ is used. A base class that holds the basic persistent object attributes and methods. To achieve persistence, the user derives a class from the persistent base class

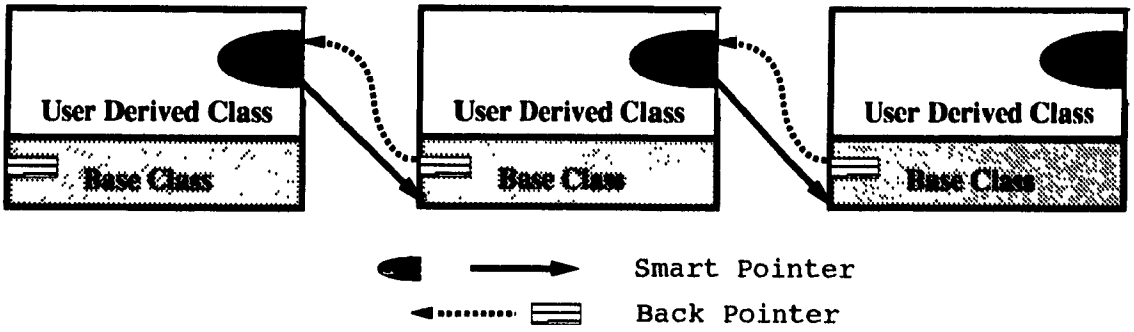


Figure 3.1: User defined "simply linked list" class using Base class and Smart pointers

and uses smart pointers in its derived class when referring to other persistent objects. Fig 3.1 shows the relation between persistent base class, smart pointers and user classes.

3.4 Loaded Objects Dictionary

An object is loaded from disk on the first reference to it through a smart pointer. When other unswizzled smart pointers reference the same object, they must locate this object in memory in order to avoid multiple loading of an object in the same program.

Numerous methods can be used to do this, from a linked list to a hash table. A search tree is used since it directly maps to the persistent object names (full pathname). A tree is more efficient than a linked list and its hierarchical structure allows for easy interactive browsing through the loaded objects.

The loaded object tree is made of three classes, Root, Dir and PersistentObj. The PersistentObj class is the base class used to attain persistence. PersistentObj instances are always leaves in the tree. The tree grows gradually as objects are loaded in memory. Each component in the object name (pathname) represents either a parent directory or the object itself, for the last component. Figure 3.2 shows an example of a user defined doubly linked list class.

In the figure, the user has four objects: /A/C, /A/D, /B/E and /B/F linked together in a list. The base layer represents the loaded object tree built from its three classes: a

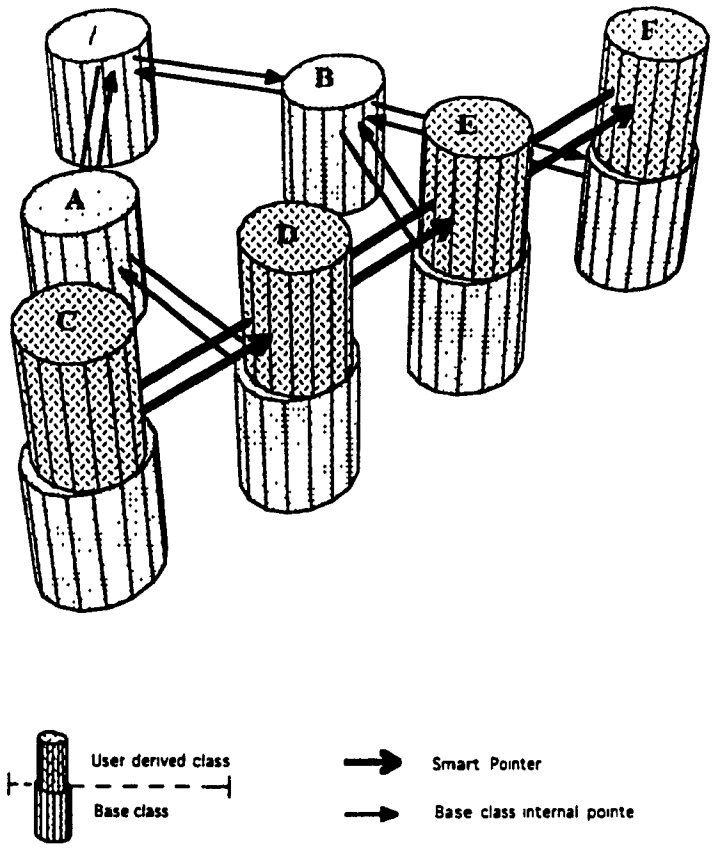


Figure 3.2: Loaded Object Tree

Root (/), two Dir (A,B) and four PersistentObj base Class (C.D,E,F). The top layer is the user derived doubly linked list class. To attain persistence, the user simply derived its class from the Persistent base class and used smart pointers in its linked list class without knowledge of the underlying mechanisms.

The hierarchical structure of the loaded object tree allows for recursive application of commands to a whole subtree simultaneously. This means that commands like unload or save could be given to all (or a subset of) the objects loaded in memory by applying this command to the tree_root object or to a given directory.

By combining this feature with the object destructor it can be guaranteed that the objects will be saved automatically at program termination, without user intervention. If all object destructors contain the "unload" command, whenever an object is about to be

destroyed, its destructor will be called and it will unload itself to disk.

The analogy with the UNIX file system may be pursued further. In particular, a mechanism similar to the UNIX mount is described in the next section.

3.5 The Mount Mechanism

The UNIX mount command allows the system to map a fit subtree to another part of the file system. When used with NFS, it does the same mapping but across a network connection to another host. It has proved to be an efficient and seamless interface to distributed storage and is now an industry standard.

In this project, a mount-like command is implemented to perform name mapping between networked client/server connections. It is strongly inspired by the UNIX mount/NFS system [65].

When a mount command is executed on a client, the client requests permission to mount a server's directory with certain read/write access. Once the permission is granted, the client writes the mounting information in the mount point.

A mount point is a directory in the local memory tree. It contains the server id, the mapped directory on the server and the access permissions. The access permissions can be read-write or read-only. Every object accessed through this mount point will be imposed at least the access restrictions. Once the mount command is issued, object names are simply mapped to their corresponding name on the server when objects are accessed. Figure 3.3 shows an example of mounting a remote server.

In figure 3.3, the client host FOO mounts "/L/N" from server host BAR on its mount point "/B". This means that on host FOO, "/B" at the beginning of any path will be replaced by "BAR:/L/N" when trying to save or load the object. For instance, when trying to load object "/B/Q/T" on host FOO, the system would automatically transform the name into "BAR:/L/N/Q/T", retrieve the object from host BAR and load it under "/B/Q/T" without the user knowing where the object was loaded from.

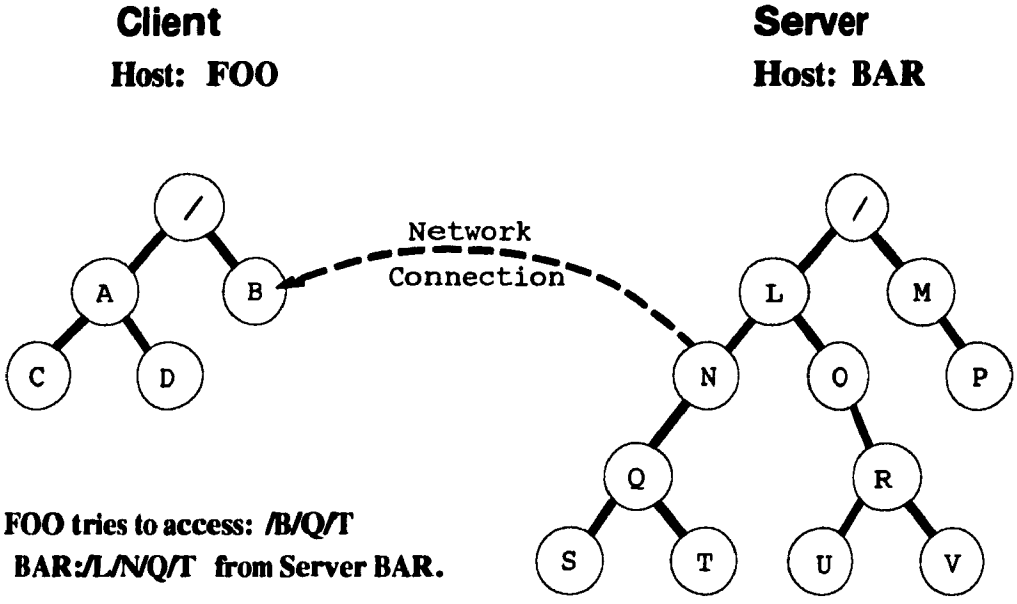


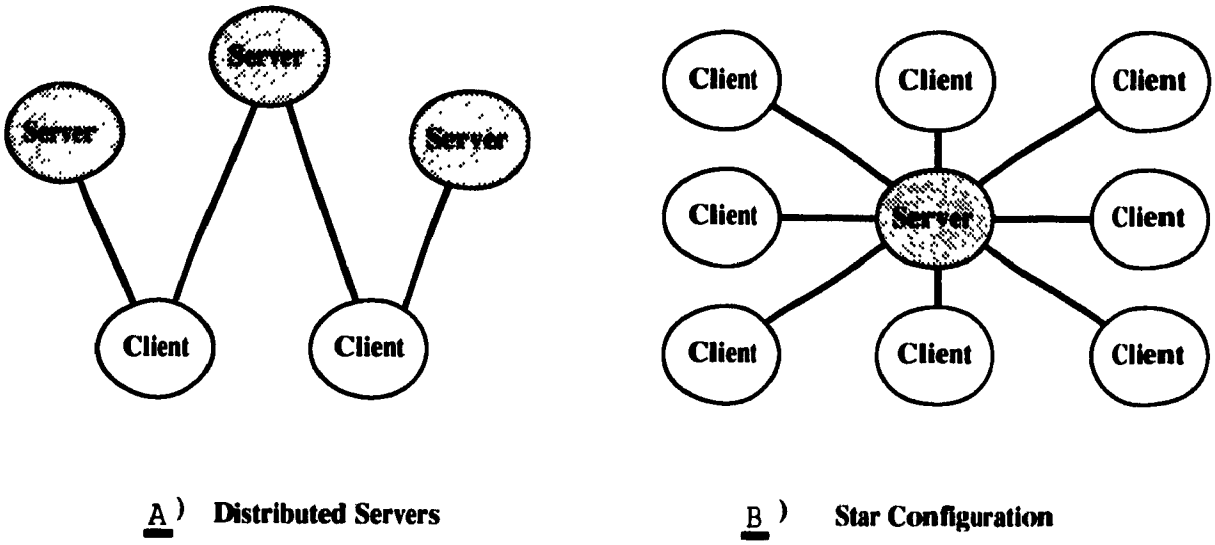
Figure 3.3: Example of mount

This provides the system with a seamless network interface. Directories are mounted from servers at the beginning of the application, and the rest is transparent. Similarly to UNIX NFS mount, many directories can be mounted from as many different servers, hence providing distributed data storage.

The distributed data storage approach allows for many clients to query many servers in various ways (Fig. 3.4-A). It has certain advantages over the more conventional "star" configuration (Fig. 3.4-B) where many clients query one central server and are limited to only this one. With distributed databases, the load is distributed across possibly many servers.

This simple name mapping mechanism enables very interesting applications. It allows for instance, mounting alternative versions of a library without any disturbance to object names.
ex:

A project references, among others, the cell "/lib/vlsi/cellnameXYZ". The



A) **Distributed Servers** B) **Star Configuration**

Figure 3.4: Client / Server configurations

"/lib" directory is normally mounted from the server's "/usr/vlsi_package/lib-1.54". A new release of the library arrives and the project's policies call for a transition period where both libraries must be available on the server so that all groups do not have to switch libraries at the same given time. Both libraries are stored side by side on the same server as:

```
/usr/vlsi_package/lib-1.54
                  /lib-1.55
```

When a project is ready to switch to the new library, all there is to do is a mount of the new "/usr/vlsi_package/lib-1.55" library instead of the old one. When all projects have successfully switched to the new library, the old one can be moved to tape archive to save disk space.

Alternatively one can use very long object identifiers that contain the host address and the time of creation. Objects across the network can be uniquely identified this way at the cost of a complex identifier. The additional level of indirection brought by the mount mechanism makes the name of the object independent from its physical location

Another major point for such a naming scheme has to do with library merging. A system using numbers (as opposed to pathnames) to uniquely identify its objects can guarantee this uniqueness only if it allocated all those numbers itself. If two teams of designers work in parallel on different servers and eventually want to merge their work in a common library, the system has to use some special merging facility to scan all the numbers in one database and compare them with all the numbers in the other database and change any conflicting identifiers. This means that every reference to the old identifiers must be found and modified to preserve the database integrity.

3.6 The Server

Permanent storage is achieved through the server via the network. This section describes briefly the principles behind the server. For a more detailed study of the server, the reader is referred to [23].

3.6.1 Server Structure

The server is mainly divided into two parts, memory tree and disk storage as shown in figure 3.5. The memory tree is similar to the one used in the client to keep track of which object is loaded.

In the server's memory tree, each node is either a directory (internal nodes) or a persistent object (the leaves). The directories store such information as who mounted a given directory and with what read/write access. The leaves contain the name of the persistent object, the list of clients that *subscribed* to it and each client's write access.

The memory tree insures proper locking, mounting, notification and all the other services offered by the database. It is implemented as a C++ class *Server* with publicly accessible methods to offer its services. To create a server process, one simply has to instantiate the class *Server*.

In order for the server to have the ability to store any complex object, it is kept

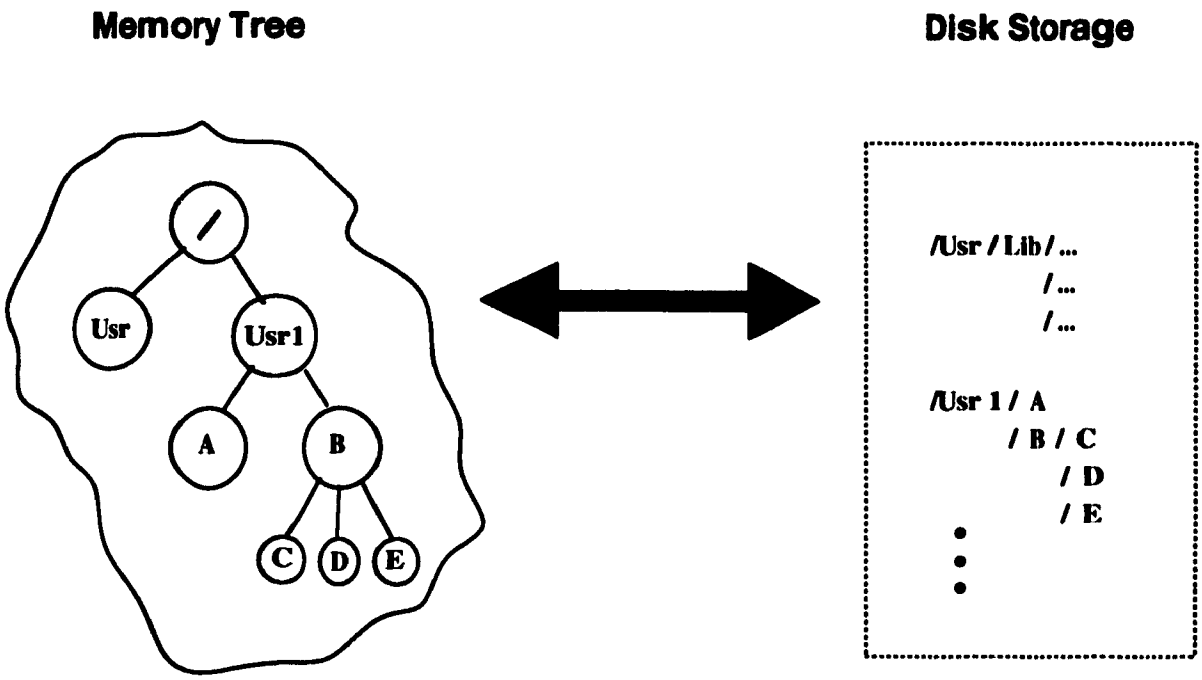


Figure 3.5: Server's logistic and storage trees

independent (ignorant) of the object's internal constituents. All knowledge of the object is kept in the client since it is the creator and user of the object. To make this possible, each object is passed to the server as a string associated with a name in the form of a pathname. The server can directly map the object name (pathname) to the UNIX file system and store the string representing the object as a file.

This method is quite appropriate for large objects but wasteful and inefficient for small objects like those used in CAD frameworks. This is why the "granularity" of the storage can be modified from one object (in a single version) per file to many objects with all their versions in one file.

Because of the simplicity of representation of persistent objects (a string with a name), the memory tree is independent of the storage mechanism. A conventional relational database could also be used as a data repository.

3.6.2 Concurrency Control

Some architectures like ORION [35] adopt a one server per client approach. A main server receives the requests and forks a lightweight process server to handle each client. This approach has some advantages among which is the guarantee that a stalled client will not interfere with the serving of other clients. It will at most stall one of the many lightweight processes while others continue to provide service. A problem with this is the increased need for locking mechanisms both at the storage level and the memory tree.

The approach used in this server is much simpler. All requests to the server are serialized (queued) to one single server process. Integrity of the memory tree is guaranteed without any locking since no two processes can modify it simultaneously.

To guarantee database integrity, only one server process can be started per database. In the current implementation, since the objects are stored in UNIX files, a database becomes a subtree on the file system. Normal file system access permissions can be used to restrict each server to a different file subtree (database).

3.6.3 Storage Policy

The system provides mechanisms to protect database integrity without using the conventional write-ahead log. It is inspired by the POSTGRES database [60] and is achieved via a "no-overwrite" policy. With this method, the old records remain in the database whenever an update occurs and serve the purpose normally performed by a write-ahead log. The updated version is differentiated from the previous ones by its timestamp.

The advantages of this approach are:

- reducing the amount of code written to support crash recovery. By keeping the history of objects, recovery is achieved by simply retrieving the last committed object version and aborting the uncommitted ones.
- Asynchronous daemons can take care of "vacuuming" obsolete records and possibly archiving them.

- supporting *undo* actions that the user may explicitly wish to apply on certain transactions.

On-disk objects are updated by database transactions. Each transaction is assigned a unique identifier. The server maintains a log file indicating the committed transactions along with the opened database files. In the event of a soft crash (when disks are not damaged), the log file is accessed and all the opened files are visited, deleting the uncommitted transactions.

The system can provide mirroring of disk objects and log file on a separate disk to allow recovery from hard crashes (when one disk is damaged).

3.7 Versioning

Each object has a version number and a timestamp associated with it. These two attributes are part of the full object name. Hence, an object named "/foo/bar/cox" of version 1.57 and timestamp 0847562 is stored as "/foo/bar/cox:1.57,0847562".

3.7.1 Versioning Strategy

The timestamp is necessary because of the "no overwrite" policy of the server. In a conventional database system, when a user makes minor modifications to an object, he only updates the object on disk (overwrites the previous version). The version number is not affected. In order to mimic this behavior and still differentiate an updated object from a previous one, the system uses the timestamp.

Having the timestamp as attribute also allows time oriented queries like: "What objects where modified during the week of June 9th?", or "When was version 4.39 of object X created?".

Since most of the transactions are performed on the most recent version (and timestamp) of the object, it would be inconvenient to always specify the full object name with version and timestamp. This is why three object naming methods can be used.

1. /foo/bar/cox:1.57,0847562
2. /foo/bar/cox:1.57
3. /foo/bar/cox

The first method is the full object name, generally used for backtracking/undo facility. The second can be used by the user to request a specific version of the object. When the timestamp is not specified, the most recent one is used as the default value. The third form is the one commonly used, where the most recent version and timestamp are used as default values. It is not possible, and would not make much sense, to specify the timestamp without the version number.

Note that this version naming convention reserves two characters: ":" and "," as version and timestamp separators. These two characters cannot be used in an object name (not even in the directory part of the pathname).

When saving an object through the server, the user has some control over the version number. He can specify any version number higher than the current one or decide to simply increment the current one. Increments can be made at two levels: a minor version increment would change version 1.57 to version 1.58; a major version increment would change 1.57 to 2.00. The user can also opt for no version change in which case the new object will only be differentiated by the timestamp. The timestamp is assigned by the server to every object written on disk and the user has no control over it.

3.7.2 Limitations and the Project Manager

Because of the server's "no overwrite" policy, only the current version and timestamp of an object can be checked out in a read/write mode. All other versions can be checked out for reading only. Because of this, only a limited set of primitives is implemented.

To provide a more versatile front end, a project manager is needed on top of the elementary versioning mechanism. As an example, consider the situation where the user

wants to get and modify and old version (or timestamp) of an object. The following steps have to be done:

1. Check out the current version in read/write mode.
2. Check out the old version to modify in read only-mode.
3. Copy the content of the old version in the current one.

After this operation, the current version is now a copy of the old one and can now be modified. It is equivalent to backtracking to the old version, after realizing an error. In a non versioned environment, it would be necessary to retrieve an old backup and overwrite the current version with it. Those steps are automated in the project manager.

The mechanism proposed here is an enhanced version of the first phase of the *two phase version* mechanism described in the previous chapter. The enhancement lies in the elementary version resolution, the ability to accept an object name without version or timestamp and get a default value.

A higher level policy oriented entity is needed to provide services like *two-phase versioning, long time locking, group access, workspaces,* and *configurations.* The version service provided here is only concerned with version numbers and data integrity by assuring proper object locking.

One facility that should be implemented at the lower level is branching and merging of versions. It is also frequently called prototyping or tentative version. Branching is splitting a version into two parallel versions of the same object. It serves to evaluate alternative designs, while development of the project continues on the main branch of the version tree as illustrated in Fig. 3.6A. Merging is done when the prototype is successful and becomes the main branch (Fig. 3.6B)

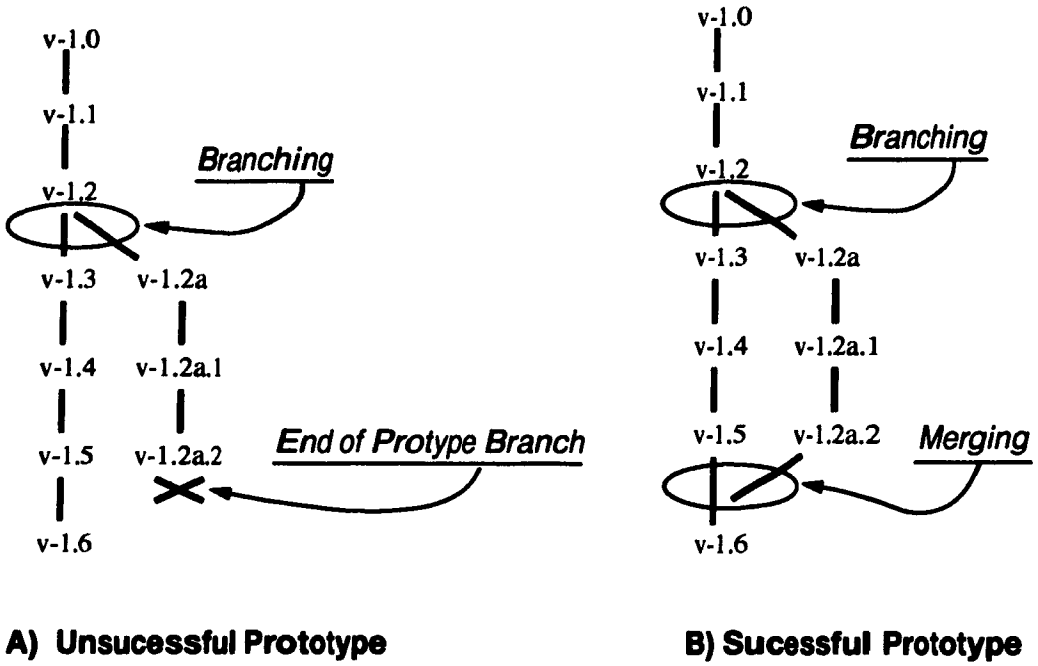


Figure 3.6: Branching of versions

3.8 Type Checking

Since the C++ language does not support persistence, it also does not support type checking of persistent objects. Compilers deal with what could be called "static type checking" where everything is checked at compilation. Persistent object systems have to deal with "dynamic type checking" since these objects are loaded at runtime.

To achieve dynamic type checking, more information than is usually provided (class hierarchy description) must be available at runtime. To circumvent that, each persistent object has a static (shared) attribute with the name of the class from which it was instantiated. The smart pointer has an attribute containing the name of the class type it points to. At load time, the smart pointer's class name attribute is compared with the one in the object to be loaded. If they match, the type is correct.

Such simple type checking would be sufficient for "C" structures but C++ classes require a more sophisticated system. As seen in a previous section, C++ allows a pointer

to a base class to reference an object from a derived class. This means that an object derived from class "ball" could be stored on a server and later be legally retrieved with a smart pointer to the parent class "circle". The simple type checking described above would fail to recognize this as a legal assignment.

The solution to this lies in an extension to the simple type checking mechanism. When the first type check fails, the system will verify if the pointers type is a parent of the object. Failing that, the object is not loaded and *type mismatch* error message is displayed.

Dynamic type checking must be performed on two occasions:

1. When loading an object from disk,
2. When swizzling a smart pointer to an object that was already loaded (found in the memory tree),

3.9 Storage of Complex Objects

To store a complex object, a special disk representation is required. As seen earlier, the most complex part of storing an object is representing the references to other objects. This is solved with the use of Smart Pointers.

With the reference problem solved, objects are composed of elementary types that can all be easily represented on disk. The disk can be accessed directly or through a networked server. The storage mechanism must be modular and independent of the target storage media. It must also be general enough to handle any complex object.

Since complex objects can have different numbers of variable length data members (attributes), each object type must take care of its external representation. Indeed, only the objects know what their internal attributes are. The network mechanism and the server must remain general enough to accommodate any kind of complex object.

Hence, data members of complex objects are serialized into an ASCII string by an appropriate method defined in each class. The ASCII representation takes more space but

allows for easy debugging and independence of endian (big or small) ordering when dealing with an heterogeneous network of workstations.

3.9.1 Serialization of a Complex Object

A complex object can be composed of many derived classes and each class knows its attributes. Figure 3.7 illustrates the serialization procedure. Each class of an object builds a substring for the complex object. All substrings are put together to form the full data string that represents the object. The final string is sent to the procedure that interfaces with the network.

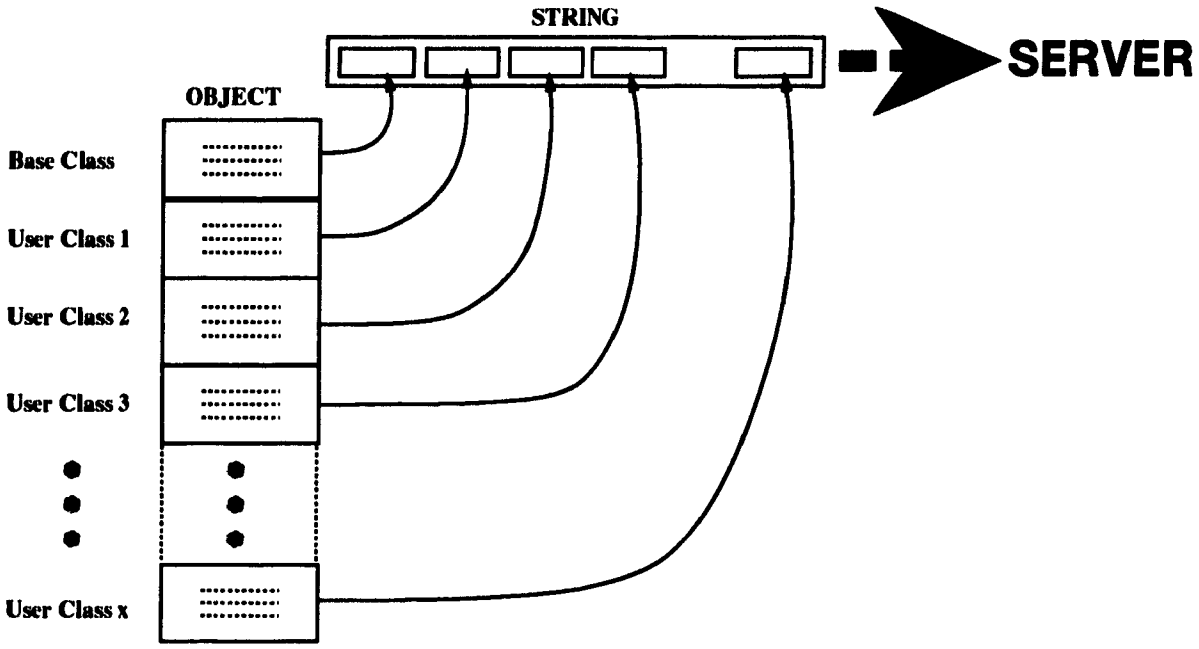


Figure 3.7: Serialization of a complex object

Figure 3.8 shows the loading process. When loading back the object, the string representing its data members is read by the client from the server. From there, each substring is read into the object by its respective class.

The encoding/decoding sequence of an object is performed with the serialize()/load() methods specific to each class.

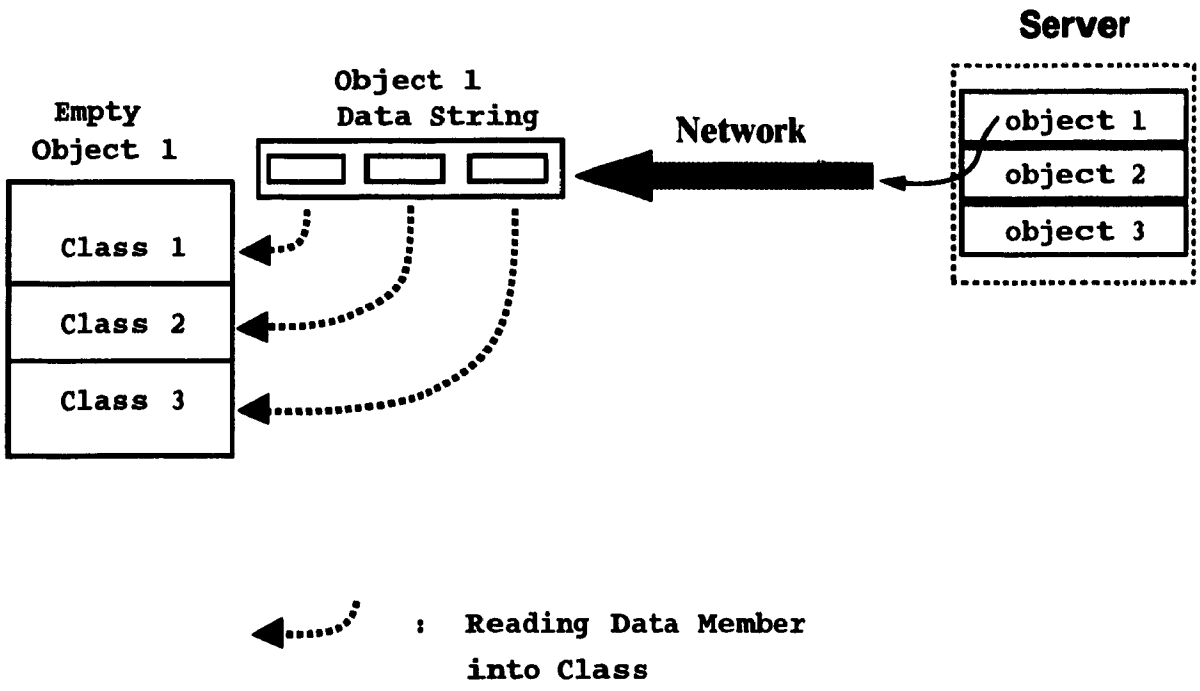


Figure 3.8: Loading a complex object from the server

3.9.2 Serialization Protocol

To create the ASCII substrings, the protocol described below is used:

Every data member is separated by a white space " " character in the string representing the object. Each data member is transformed into an ASCII representation and appended to its class' substring which in turn is appended to the main string representing the complex object.

The protocol supports the following types: int, long, short, unsigned, float, double, char, String (Class String) and Smart (Smart Pointers).

Some types like char, String and Smart cannot be directly appended to the string since they may contain white spaces. For these particular types a special encoding scheme is used.

For class String and Smart, the internal ASCII string is extracted and encoded as string

length followed by the string. The exact format is:

- STRING-LENGTH, white space(s), "@"STRING

For instance, "hello modern world" is encoded:

18 @hello modern world

and " hello modern world" (note the space before hello) is encoded:

19 @ hello modern world

This string representation is also used to encode both the substrings of each class and the full string containing the object. A single character (type char) is represented in a similar manner but with the length omitted. Thus, a char attribute holding the value "t" will be represented as "@t" in the string and a " " will be represented as "@ ".

3.9.3 Class Conversion

Objects are stored on the server and may remain there for an arbitrary period of time. Between the time it was stored and the time it is restored, the user might have modified the class defining the object.

When the set of attributes of a class is modified, the serialize()/load() methods are modified accordingly to enable proper encoding and decoding of an object between the server and memory representations. Because the encoding/decoding sequence for the object has been modified, it will not be able to recognize objects stored using a previous version of these classes.

One method to correct this situation is to apply a filter to the entire database to convert all objects on disk from the previous version to the current one. The complexity of such an operation makes it prohibitive. The class builders do not necessarily control the server, and may not know the server's storage representation.

A second method, adopted here, is to automatically translate older versions at load time. To allow this, each class prepends its name and version to the substrings it generates when serializing an object. When a modification is made to a persistent class, the version of the class is incremented and a translator between the previous version and the new one is written. At load time, the class reads the substring and looks at the version number. If it corresponds to the current one, the object is read directly. If it is an older version, the translator is called and the object is loaded under the new format. When the object is later unloaded from memory, it is saved under the new format.

This method is transparent to the user and requires little work from the class builder. It is also versatile in the sense that a translator is not applied to the whole object but rather to the specific class that was modified. This simplifies the maintenance and upgrade of packages that use the persistent store.

3.10 Networking

The proposed system is designed to work in a local area network (LAN) of UNIX workstations. In fact, any client can access any server anywhere on the Internet [14]. Performance, however, dictates usage within the bounds of a LAN.

Workstations with a BSD (Berkeley University) derived UNIX traditionally communicate with each other through *sockets* [58]. The socket interface treats network connections as files and uses the traditional functions: "open, creat, close, read, write", which is very natural to programmers. A newer networking interface named Transport Level Interface (TLI) [65] is promoted by UNIX Sys-V.4 and provides similar services but with a somewhat different set of commands.

Both sockets and TLI use a selectable underlying communication protocol. Traditionally, User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) running on top of the Internet Protocol (IP) have been used for connectionless and connection-oriented protocols respectively. Newer protocols are being developed which match more closely to the OSI model [66].

A higher level alternative to sockets and TLI is the Remote Procedure Call (RPC) interface. The latter allows a function to be executed on a remote computer while presenting exactly the same interface as if it was a regular local function. RPC uses either sockets or TLI as an underlying transport mechanism, hence making it more portable across different platforms.

Ideally, a network interface should be modular enough to prevent code dependency on the underlying protocol. It should be portable on many platforms and easy to modify or replace. RPC being a higher level interface would seem like the ideal choice. However, RPC is most suited for stateless servers that do their job and die immediately after. This system requires a long lived server. Moreover, the notification mechanism requires greater flexibility than what RPC can easily accommodate. For these reasons, a socket based interface is used.

Special care is taken to regroup any dependency on the network interface to a small number of functions. If the need is felt to port the system to a different networking interface, only this restricted set of functions would have to be rewritten. Even then, thanks to the serialization of objects, the network interface is limited to simple exchanges.

3.11 Notification

Earlier in this document notification was mentioned without much detail. There are two kinds of notification of interest to us.

The first, *data notification*, is what is usually called notification. Data notification is the action of notifying a process that some data of interest has been modified. The second, *method notification* is the action of asking an external process to execute one of its methods of a certain object. *method notification* resembles the RPC mechanism with an added level of indirection, as explained later in this section. The following example will illustrate these two concepts.

In a VLSI CAD framework, the designer uses the graphics editor to modify some part of the design. It simultaneously uses an incremental critical path evaluator to find the

bottleneck in his design. As he modifies the design, the graphics editor notifies the critical path evaluator of what objects were modified, hence *data notification*. The latter will find the critical path and tell the graphics editor to highlight the objects making the critical path. In other words, it is going to request the graphics editor to execute the **highlight** method on a list of objects forming the critical path, hence *method notification*.

In other CAD frameworks, data notification is made easy by keeping all loaded objects in a single memory location and manipulating them there (Oct). This is done at the cost of greater limitations on the size of simultaneous designs on a single server (the server has a limited amount of memory). In most of those designs, method notification is done in a tightly bound manner and usually only to the graphics editor.

What is presented here is a general purpose notification mechanism. One approach is to have each tool communicate with each other and provide an RPC library of the functions for use by other tools. This presents a few problems. First, the programming overhead in each tool would be considerable and much of the functionality would have to be replicated in each tool. Second, this would break the network and persistence transparency mechanism since each tool would have to know the location of the other tools on the network.

A second more generic approach is to treat both data and method notification as *event notification* without distinction between the two. In the persistence mechanism presented, the object is transparently fetched from the server to the requesting client. As an extension of this method, when the client registers for an object with the server, it can ask to be notified of any change made to that object. If the object is changed, the server sends a notification message to the client.

With the approach of "notification through the server", the tools do not need to be aware of each other's location or identity since everything is received and dispatched by the server. Moreover, in such a centralized notification scheme, if functionality like authentication is later added, the modifications can be limited to the server as opposed to reworking all the tools.

Since only one client can register for an object in *read-write* access mode, all others can either have *read-only* or *read-notify* mode. The server keeps a list of clients accessing each

object along with the access mode. With this list the server can provide multiple client notification.

Naturally, such a scheme requires cooperation from the client with *read-write* access on an object. If such a client modifies the object, it must eventually notify the server. This has an advantage over the "spying over the shoulder" or "polling" method used in other systems. Since the *modifying* client only lets the server know about the modification when it wants to, it can choose to commit modifications only at milestone modifications.

Method notification is provided in a similar centralized way. When a client registers for objects, it can also make available a list of methods accessible by other tools on those objects. This action is called *exporting* methods. Other clients of those objects can then notify the server that they want a given method executed on a given object. The server will in turn notify the clients that made this method available. This implies the possibility to perform method notification on many clients simultaneously.

Naturally, method notification implies that tools using method notification must know a bit more about the other tools than simple data notification. They must know what methods are exported by other tools in order to use them. This requirement is common to any method notification system.

As in the data notification, the added level of indirection provided by the server eliminates the need to know the location of other tools on the network as encountered if direct RPC were used.

A useful example of this would be a class tutorial on CAD frameworks where the instructor does a design example and each student only runs a graphical editor with all its methods exported. The instructor's program could then make notifications for each action its graphic editor performs, hence providing an "exact dynamic copy" of the operation to the students.

As for data distribution, clients can perform transactions with many servers. Notification mechanisms are associated with the objects, hence associated with the server storing each object. The server deals with all the physical locations concerns and the multiple distribution of notification messages.

The major improvements over existing methods are:

- Data and method notification in a networked framework,
- No limitation on the type of objects or methods to notify,
- Modular and uniform programming interface for all notifications,
- Independence of the physical location of different tools.

Chapter 4

Implementation

This chapter will detail the implementation of the concepts explained in the previous chapter. Most of the elements presented here have been implemented in a prototype. The latter is evaluated in the next chapter.

4.1 Architecture

This section studies the main data structures (classes), and their hierarchy, used in this project. The heart of the system revolves around two structures: the *Smart Pointer* and the *Loaded Object Tree.*

4.1.1 The Loaded Object Tree

The loaded object tree is based on four classes of objects: Object, Dir, Root and PersistentObj. Class *Object* is the base class from which all other objects of the tree are derived. From that class, two branches are created. The first is *PersistentObj* used as base class for persistent objects defined by the user. Objects instantiated from class PersistentObj are the leaves of the tree. The second class derived from *Object* is *Dir.* Dir objects represent the nodes of the tree, holding it together. Only one node is different from the others in

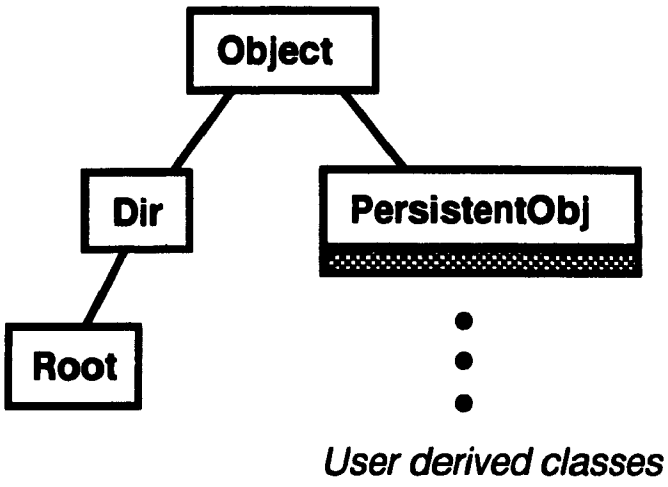


Figure 4.1: Class Hierarchy of Loaded Object Tree

the tree, it is the root node. It is instantiated from class *Root* and is unique in the tree. Figure 4.1 shows the class hierarchy forming the loaded object tree.

In the "*loaded object* tree", the names of the objects are mapped on the internal nodes and the leaves of the tree (Dir and PersistentObj). Figure 4.2 illustrates how each token of the object's full pathname is mapped on a node of the tree.

No single object in the tree contains its full pathname. The real objects are the PersistentObjs and to obtain their fully qualified name, they have to walk up the tree to the root. The tree has the functionality to reconstruct a pathname when needed, answer queries for pathnames as well as other pathname manipulation operations required to manage the contained objects. Much of this functionality is spread throughout the class hierarchy forming the object tree.

Class Object

Class Object is the base class of the object tree. It contains attributes and methods common to all its derived classes. The basic attributes are:

- **name:** One token of the full pathname.

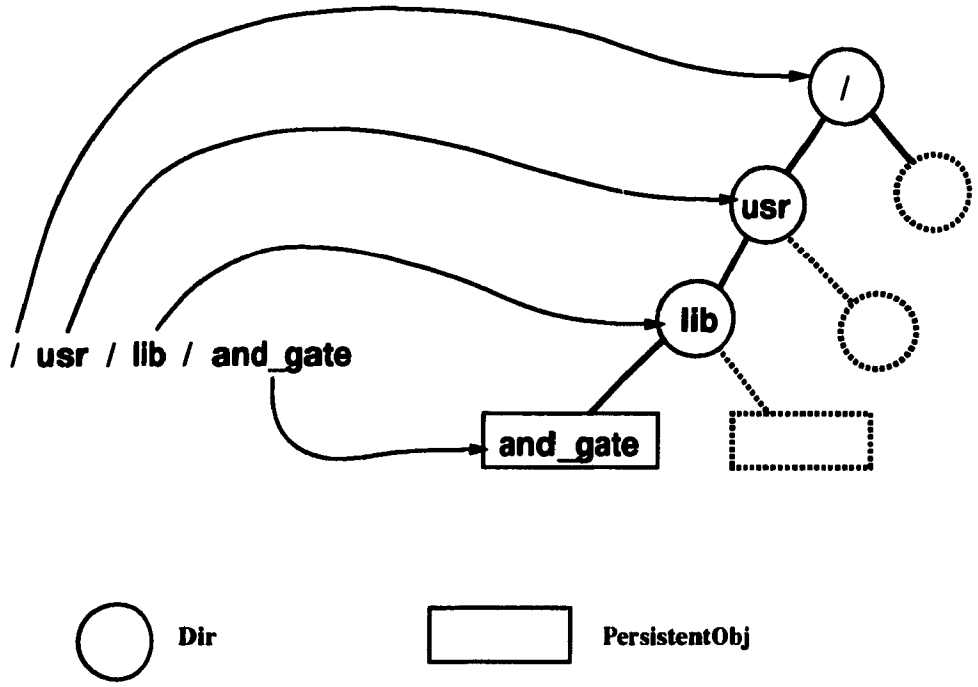


Figure 4.2: Mapping of a pathname on the Loaded Object Tree

- **version:** major and minor version number (ex:1.57).
- **timestamp:** Timestamp assigned by the server.
- **parent:** Pointer to parent Dir in the tree.
- **current status:** Is this the current version of the object?

Most of those attribute names are self explanatory. All objects in the tree have a **parent** Dir except for the root. The **current** status field is used to identify the current version of the object when it is loaded.

Some functions exist in both class PersistentObj and class Dir performing the same basic operation but on different classes. In tree manipulations, a basic pointer to class Object is used to reference both Dir and PersistentObj. To allow access to functions defined for those classes with this type of pointer, the functions have to be defined as

virtual or (pure virtual) in the base class Object [22]. Most of the functions in classes Dir and PersistentObj are virtual.

Class Dir

Class Dir is derived from class Object. By deriving from class Object, it inherits all of Object's attributes and methods, including a name and a version.

A Dir contains the following attributes:

- **mount type:** local or remote mount.
- **mount host:** Server to which it mounts.
- **mount dir:** Server Directory to be mounted here.
- **mount flags:** Is the directory mounted in a read-only or read-write mode?
- **default unload flag:** Action to take at unloading time.
- **son list:** List of sons (leaves or subtrees).

A Dir object can either be a mount point or a regular Dir. Since any Dir object can become a mount point upon execution of the mount command by the user, the basic attributes must be present in each Dir object whether it is used or not. Those attributes identify the server to which it connects, the directory on the server that will be mapped to it and the write access flags of the mount.

The current implementation simply uses the name of the workstation hosting the server process as mount host. This is enough for the purpose of prototyping but should be refined to allow many server processes to run on a single host. A **mount type** field exists in the Dir class to allow for future implementation of a local server.

For now, only remote servers are supported. In a normal design scenario, a designer will share some objects with a workgroup. These shared objects should be on a networked server (remote server). However, some of the designer's work could be private and the

resulting objects stored somewhere on a local disk. In this case, it would be more efficient to have a private server process linked with the client tool for local private access. This local server would then have direct internal connections with the client process, hence speeding client/server transaction.

Class Dir has a **son list** attribute to keep track of its sons. Since the object tree is an *n-ary* tree, no fixed number of attributes can be reserved to hold pointers to children. A dynamic list must be used to keep track of the sons.

Typically, the degree of any node is not very large and a simple linked list implementation is adequate to store a list of pointers to the sons. The list is encapsulated in a class *Son_list* to allow transparent modification to the internal structure without affecting the tree mechanism. Such modifications could be changing from a linked list to a more efficient hash table [36] [16] if the number of sons increased significantly. Class Son_list also encapsulates methods to search for a given son, or to list the sons. The elementary version-completion mechanism discussed in previous chapters is implemented in this class.

A Dir object can have two types of sons: a subtree (Dir) or a leaf (PersistentObj). To allow both types to be referenced, class Son_list uses a pointer to type Object. Finally, to speed up searches, the tree only keeps track of the objects that are loaded in memory.

Finally the default unload behavior attribute will be discussed in a later section.

Class Root

Class Root does not contain any attributes. It is used to provide a different behavior for some of the internal tree manipulation functions. One such function is the rebuilding of a full path name. It knows to stop when it reaches the Root object.

Class PersistentObj

Class PersistentObj is the base class used to attain persistence. It inherits the methods and attributes of class Object and adds the following attributes:

- **back pointer list:** List of pointers to swizzled Smart pointers referencing it.
- **unload behavior:** Action to take at unload time.

The **back pointer list** is used to keep track of the swizzled smart pointers referencing the persistent object. Such a list is necessary to preserve the integrity of the smart pointers when loading/unloading persistent objects. This list is encapsulated in a class similar to Son_list. This class also contains the mechanism to add, remove or browse through the list. The "remove" method is called when a smart pointer is deleted to update the back pointer list accordingly.

The **unload behavior** attribute is used when unloading an object. It will be further discussed in a later section on loading/unloading of persistent objects.

4.1.2 The Smart Pointer

Figure 4.3 shows the class hierarchy used for the smart pointers. It is implemented as a general base class and a parameterized derived class.

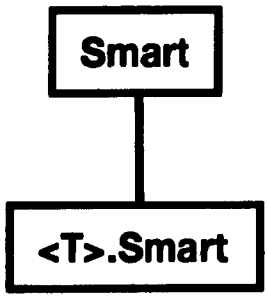


Figure 4.3: Class Hierarchy of the Smart pointers

Smart Pointer Hierarchy

Parameterized smart pointers are required for dynamic type checking of loaded objects. A base class "Smart" is defined to provide a common pointer type for all Smart pointers. This way, the back pointer list for any "smart pointer" pointer can be of type (Smart *).

The base class *Smart* only contains an attribute for the name, basic methods and pure virtual functions. The parameterized class is class <T>Smart where <T> is the type of object the Smart pointer references.

Smart Pointer transparency

In order to use a smart pointer as a regular pointer (i.e. transparent usage of smart pointer), the smart pointer class overloads the operators commonly used with regular pointers. An operator is overloaded to accept different arguments.

The following operators are overloaded in relation to:

SmartPtrToTypeFoo A, B; Foo * C;

- A->...;
- A = B;
- A = "full pathname of an object";
- &A
- A&
- A == B
- C == A
- C = A

Since the Smart pointer is meant to be used as a regular pointer, the operator mostly used is "->". It provides transparent loading of the object into memory and regular member access to the loaded persistent object. Operator "=" is overloaded to prevent member wise copy of a smart pointer to another. A smart pointer copy must either make the copied pointer unswizzled or add the copied pointer to the object's back pointer list to assure

integrity of the object tree. The mechanism used here is to make an unswizzled copy of the smart pointer. The "=" operator with a string on the right hand side (rhs) is taken care of by an overloaded constructor. It has the effect of initializing the smart pointer to the string on the right hand side. Naturally, the smart pointer is initially unswizzled. If the smart pointer was already initialized before the assignment, the destructor is first called. The destructor unswizzles the smart pointer before destroying it, hence preserving object tree integrity. Smart pointer comparison (A == B) is done on the name part only, therefore making the comparison valid even if both pointers are not in the same swizzle state. Comparison and assignment with regular pointers requires another overloading of the (==) and (=) operators. Taking the address of a smart pointer (&A) and making a reference to a smart pointer (A&) is normally handled by the language.

To use a smart pointer, the user must initialize the smart pointers with the full path-name of the object of interest. The object will be loaded transparently at the first reference made to it through the smart pointer. Persistent objects should not be created manually. They may be statically scoped (local or global variable) or dynamically scoped (allocated with *new*) in the program.

The user can obtain the memory address of a persistent object and access it without the use of smart pointers. The use of a regular pointer is sometimes needed since using a smart pointer as the "index" in a graph traversal would result in poor performance.

To illustrate this statement, consider the traversal of a singly linked list where the persistent objects have a member "next" which is a smart pointer to the next element in the list. To traverse the list a smart pointer "index" is used. Every time "next" is assigned to "index" the destructor of "index" is called and an unswizzled copy of the pointer is made. When an access is made to the object referenced by "index" a swizzling operation is triggered. All this is too much overhead for time critical graph traversal. For this reason the automatic translator (=) is provided that will assign a memory address to a regular pointer from a smart pointer.

While the use of regular pointer is sometimes needed, the system cannot guarantee the validity of such a pointer since objects can be loaded, unloaded and reloaded transparently. To prevent any pointer corruption, the user should use a semaphore during the traversal

of a graph of persistent objects linked by smart pointers.

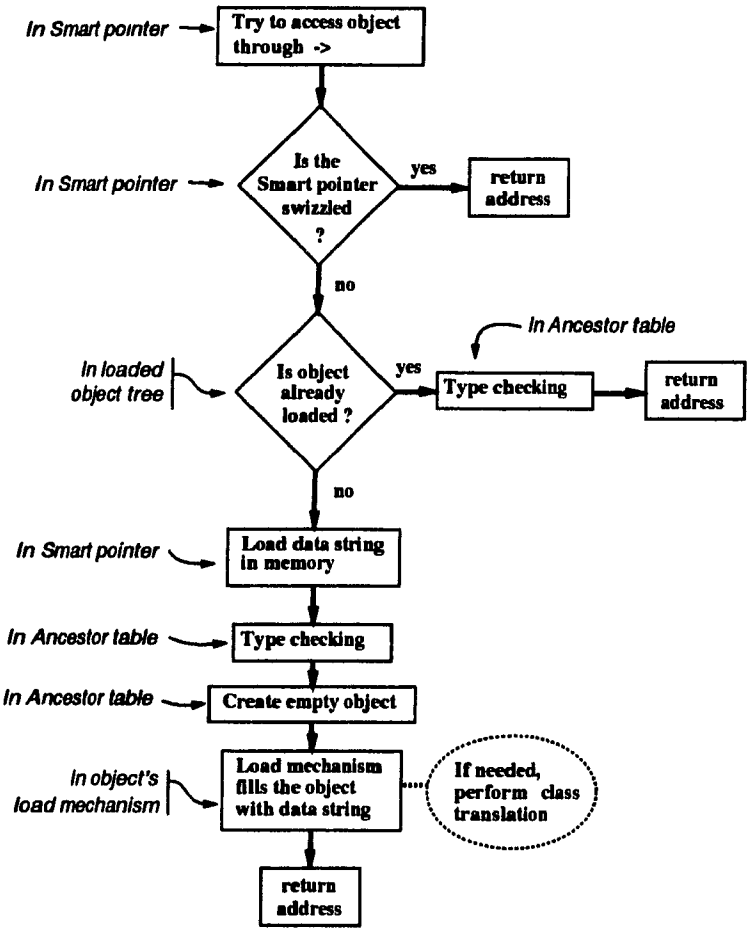


Figure 4.4: Swizzling of a Smart pointer

With the above operators overloaded, and the smart pointer initialized with the path-name of the object, the use of a smart pointer should be identical to the use of regular pointers.

The swizzling mechanism that transparently loads the object is the most important concept of the Smart pointer. Figure 4.4 illustrates the algorithm of the swizzling process including the load mechanism.

4.2 The Serialization Process and Protocol

Before going into the details of loading a persistent object, the protocol for serializing an object must be studied [37].

Figure 4.5 illustrates the calling diagram of the main functions which interact when serializing/unserializing an object. Different levels of action are followed by each module. The ***toS** and **Sto*** functions are the low level serialization routines explained in the previous chapter. The "*" in their name is replaced by the type of object to convert. ex: serializing an int is done by "itoS" and "Stoi" transforms the string representation back to an int. Those primitives are used by serialize() and load() which hold the storing/loading strategy associated with each class.

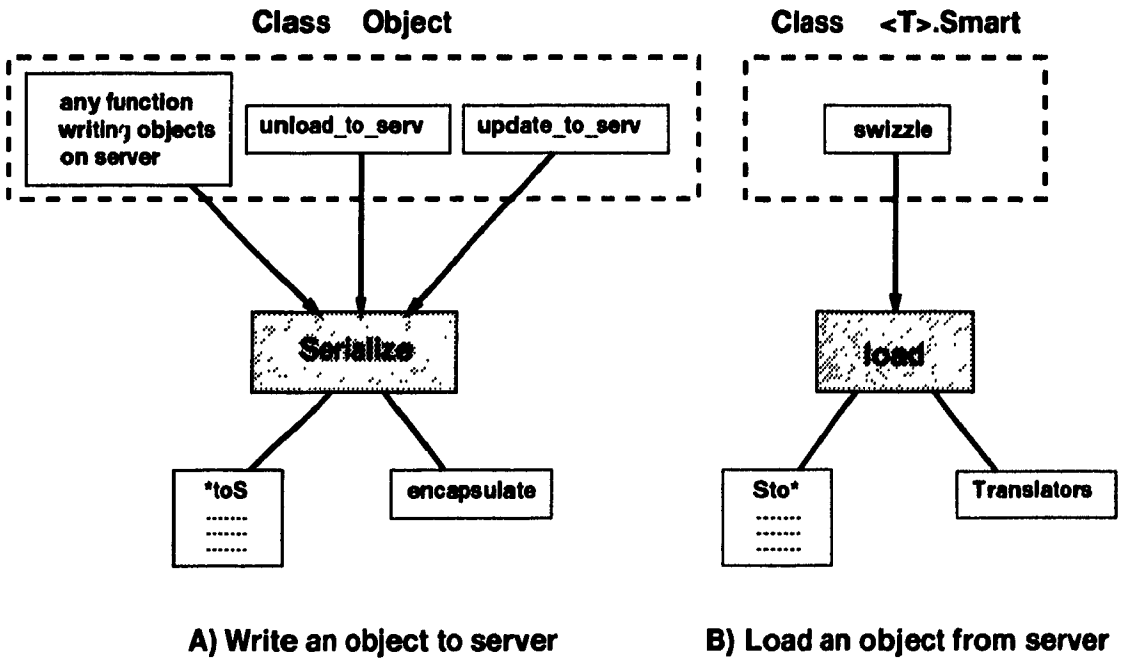


Figure 4.5: Serialization Architecture

4.2.1 Serialization of an object

Serialize as seen in fig.4.5A is called by all functions that want to write an object to the server. Those functions are: update_to_serv(), unload_to_serv() and are located in the base class Object.

As explained in the previous chapter, an object can inherit many classes where each class has a set of attributes. A whole object is stored in a string that is composed of substrings. Each substring represents a subinstance in the object. The serialize function is a virtual function redefined in each class that make the persistent object. Each serialize function knows only the attributes of its class and how to transform them into a substring. Each will in turn add its substring to the object's global string.

For instance, if class employee has three attributes:

```
int    emp_number;
String name;
float salary;
```

The serialize function for it will look like

```
itoS(emp_number);
strtoS(name);
ftoS(salary),
```

The knowledge of the attributes. their type and their relative order is hard coded in the serialize function.

Also, (not illustrated in the previous example), each serialize function must put its class name and version at the beginning of the substring for type checking and version translation purposes. To facilitate the dynamic type checking at load time, the most derived class of the object appears first in the string. This way, when the string is read into memory from the server, the first field to be read will be the class name, type checking can then proceed without further parsing of the string.

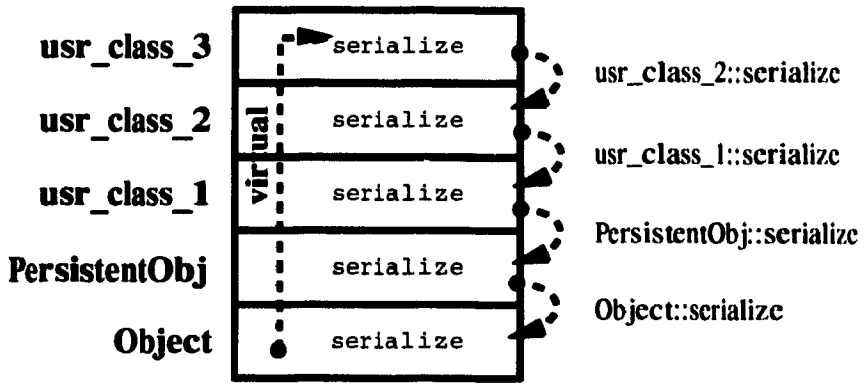


Figure 4.6: Virtual serialize function

To make the most derived class execute "serialize" first, the serialize function is declared virtual in the base class Object, as illustrated in figure 4.6. Each serialize function, after serializing its attributes, will call its parent's serialize function with a fully qualified name. This calling chain will stop at base class Object when the full object is translated to a string.

Encapsulation

The substring produced by serializing one class of an object is encapsulated and assembled with the other substrings to form the complete string representing the object. Encapsulation of each substring is done as follows:

```
|__A__|_B_|___________C____________|

\____________      _______________ /
             \    /
              D
```

```
Where: D is the encapsulated substring
       A is the length of D in in ASCII representation
```

```
C is the substring to encapsulate
B is a white space separating A and C
```

The encapsulation adds the length of the encapsulation itself before the substring. Having the length as the first field of each substring allows resynchronization in case the loading mechanism fails for a given class. Indeed, if the erroneous load function reads too much data or not enough, the next load function still gets the correct offset in the string.

4.2.2 Loading the object

Figure 4.5B illustrates how the Smart pointer, through the swizzle mechanism, loads the string into the object. This must not be confused with the higher level loading mechanism where type checking is performed. This is simply object filling, performed after the higher level manipulations studied later in this chapter.

The load function is handed a string representing the object's attributes and a pointer to an empty object. The only initialized attributes in the object at this time are its name, version number and timestamp. Load must then perform the exact reverse steps done by serialize.

Like serialize, load is virtual in order to have the most derived class load its data first. Loading of individual attributes is done by performing a **Sto*** for each attribute. Once a class is filled, it calls its parent's load function using a fully qualified name and giving it the correct offset in the global string of the object.

Translators

Since the first data to be put in each substring by the serialize functions is the class name and version, decisions can be made on how to load the rest of the substring.

If the string represents an older version of the class, a translator can be called to correctly load the old format in the new object. Figure 4.7 illustrates the translation mechanism.

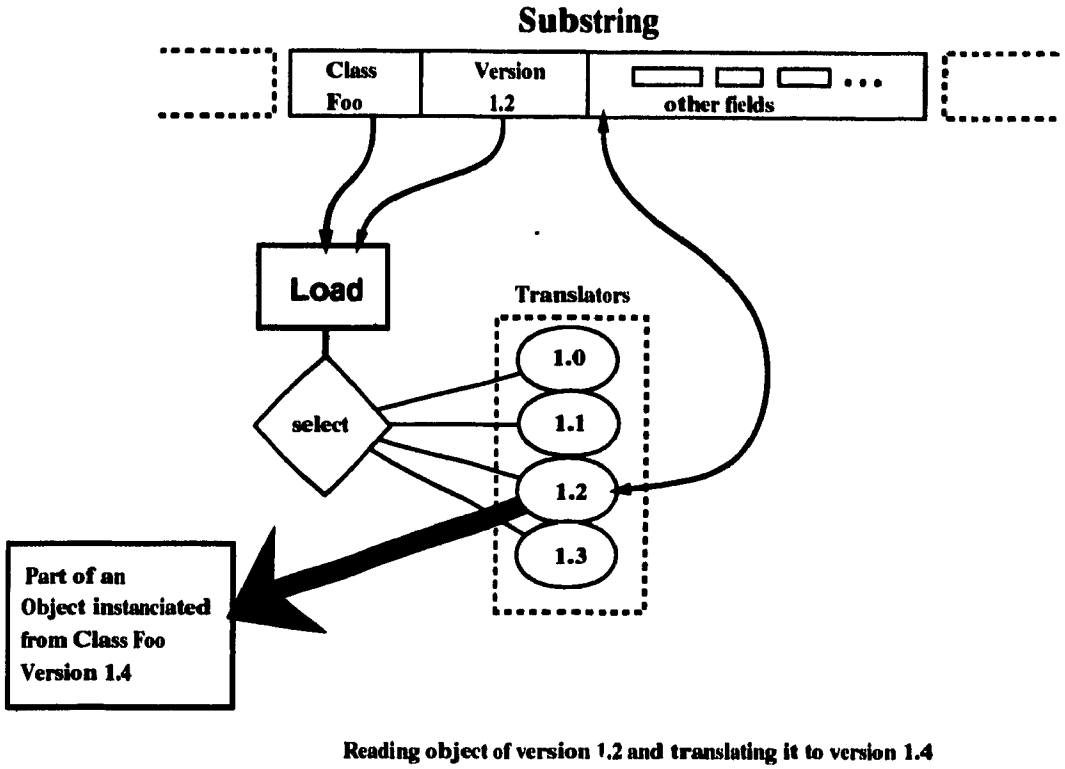


Figure 4.7: Class Translation Mechanism

The load mechanism has a translation selector based on the version number of the class to translate. Once translated and loaded, the object is in the latest format. If the object is saved to disk, it will be saved in the new format hence making the format translation permanent. This method makes translation transparent to the users and easy to implement for the designers of the classes. Note that the version number of a class is not in any way related to the version number of the object that the user is working with.

4.3 The Loading and Type Checking Mechanisms

The loading of an object is done via an unswizzled smart pointer. Before loading from the server the smart pointer checks in the loaded object tree that the object is not already loaded. If it is, it just swizzles itself with the memory address of the object. If it is not, it

loads the object from the server.

The loading mechanism is divided in four steps:

1. getting the object string (data string) from the server,
2. performing a type checking between the smart pointer and the data string,
3. creating an empty object to host the data string,
4. filling the empty object with the data string.

The first step involves looking up the loaded object tree to get information on the mount point. It needs the server id (host name) and the object name, once mapped with the mounted directory, to submit the request to the server. The read-write accesses are set in the smart pointer or defaults are used. The server returns a data string with a name, version and timestamp uniquely identifying the object.

The second and the third steps are done through an ancestor table described in the next section. The fourth step was explained in the previous section.

4.3.1 The ancestor table

The ancestor table is a mechanism to provide dynamic type checking of persistent objects in a C++ environment. The C++ language with the inheritance mechanism adds more complexity to type checking. As seen in the previous chapter, a pointer to an ancestor of an object can also be a legal pointer to this object.

A persistence mechanism must provide the same level of type checking and be as versatile as the host language. To provide such type checking, a simple name matching is not enough. The system needs an "ancestor" table to keep track of the legal ancestors of a given class. A line in the ancestor table would look like this:

- CLASS NAME, POINTER TO CONSTRUCTOR, ANCESTOR LIST

Where the first field is the name of the class, the second field is a pointer to the constructor of that class and the third field is a list of the class names of direct ancestors (the latest version of C++ supports multiple inheritance).

To perform type checking, the system extracts the type from the data string, finds it in the ancestor table, and from there recursively searches the ancestor list looking for the smart pointer's type. If it reaches the *PersistentObj* type without finding the smart pointer's type, it means that this object cannot legally be assigned to this smart pointer. In other words, the object is not an instance of a class derived from the smart pointer's type and therefore is an illegal assignment.

Another purpose for the ancestor table is to build an empty object of the correct type to host the data string. The object type leads to the corresponding entry in the table which contains a pointer to the proper constructor.

Ancestor table implementation

Ideally the C++ language should maintain information at run time about the class hierarchy. Failing that, a preprocessor could automatically extract the information required to build the ancestor table. The alternative presented here is to define an object of class **Meta** for each class in the program. The constructor for **Meta** adds the entry (**info**) for the class in the ancestor table.

```
Class Meta {   // Used to build the ancestor table.
static * info ancestor_table;  // Pointer beginning of table.
Meta(string, string, void *);
~Meta() {rm_from_table()};
}
```

```
Class info {  // Actual entry in the table.
string name;        // Class name.
```

```
string ancestors;   // List of ancestors
void * constructor; // Ptr to constructor of class.
info * next;        // Next entry in the table.
}
```

Example user class:

```
Class foo   public bar, public cox {
- - -
- - -
- - -
}
```

Line to be automatically added by preprocessor or a script right after the class definition:

```
Meta foo_meta(''foo'', ''bar cox'', \&foo::foo);
```

In this example, the class is derived from two ancestors: bar and cox. The added line creates a global object of type **Meta**. It's constructor will add an object of class **Info** in the ancestor table.

4.4 Versioning

The versioning follows the protocol described in the previous chapter. The version completion mechanism is located in the *Son_list* class previously described.

Versioning is divided into three levels of specification: name, version and timestamp. Whenever a level is missing, the system defaults to the current (most recent) item at that level.

Since the loading of objects is done through smart pointers, the object name specified in the smart pointer determines which version will be accessed. The loaded object tree has a version completion semantic identical to the server. Hence, a smart pointer initialized with just the name of the object will get the current version. When it queries the loaded object tree to see if the object is loaded, it will ask for the current version.

When the server returns the data string to the client, it returns it with the full object name, version and timestamp, even if the request was made for the current object. This allows the system to load the full information in memory. When the client saves the object to the server, it can specify a version number or a relative increment. It cannot, however, specify the timestamp field since it is automatically assigned by the server based on its own clock.

Many smart pointers can be initialized to the same version of an object. The first one to load the object will query the server. Since loaded objects contain their version and timestamp, they can all be differentiated from one another but a flag is required to specify if an object is the most recent version available. This flag is stored in the **cur_status** and can either indicate a current version, a current timestamp or simply that this is not the current object in any way.

4.5 Unloading Objects

Objects loaded in memory can be specifically *deleted* (remove the disk copy as well), *trashed* (do not save the modification to disk) or *saved* (saved to disk when unloaded). Unloading occurs when explicitly requested by the user, when an object subtree is unmounted or upon normal program termination. In future development a paging system could be designed that would allow the user to load persistent object graphs that are larger than the machine's memory.

If the system was dealing with a regular database server it could simply issue a save command from the root of the tree (for a global unload) or at the root of a particular subtree (before unmounting the subtree) and all the objects would be saved to the server.

Both modified and unmodified objects would overwrite the old ones on the server.

Because of the server's "no-overwrite" policy, this approach would be very wasteful. Indeed, in many cases, a CAD work session only modifies a small percentage of the design objects loaded in memory. To avoid such a waste, the system should try to identify the modified objects and only save those.

To provide a smart unload mechanism, each object's unload method will look up the object's **unload_behavior flag** to know whether to **save** the changes or **trash** the object. If the flag is not set explicitly for an object, the unload method walks up the tree to the mount point and looks at a **default unload behavior** flag for the subtree. This flag is set as a subtree mount option.

Each object's destructor contains the unload function which is automatically called upon unloading. However, at program termination, nothing guarantees that the leaves of the tree (the PersistentObj) are destroyed before the internal nodes (the Dirs). To unload themselves, the persistent objects need all their ancestors up to the mount point in order to rebuild their full pathname and get the server's identification.

To insure that persistent objects are unloaded first at program termination, the classes Dir and Root have special destructors. These destructors simply contain the unload command which when applied to a Dir object will recursively apply the command on all its children before applying it to itself (post-order traversal). Thus the unload commands processes the leaves first [16] [36].

4.6 Automating persistence

This section first summarizes the steps required to attain persistence and then suggests methods to automate those steps.

4.6.1 How to attain persistence

First, persistent objects must inherit from class **PersistentObj** and Smart pointers must be used to refer to those objects. There must be one and only one instance of class Root for the memory tree and it must be a global variable.

Each class must provide two functions:

1. get_class_name(): Returns the class name in an ascii string.
2. get_class_version(): Returns the class version number in a float.

Those functions are required for serializing an object into a string for storage on the server and are used for type checking at load time.

After every user class definition, there should be an instance of class *Meta* initialized with the class name, the list of ancestors and a pointer to the constructor of this class. This instance of Meta is used to automatically build the ancestor table.

Every user class must have at least one constructor that does not have arguments. Other constructors with arguments can be defined as well. The constructor without arguments is needed by the ancestor table when building the empty object before filling with the data string from the server.

Finally, the serialize and load functions must exist in every user defined class. The first function serializes the data members of the objects into a string to store on the server. The second takes the incoming data string from the server and fills the empty object with it.

4.6.2 The preprocessor

The first two steps, using smart pointers and deriving the user objects from class PersistentObj, are done manually since they imply the reorganization of an existing program or simple programming practices when developing a new package.

The other steps can easily be automated with a pre-processor that would generate the required code.

The instantiation of the Root object is done automatically by linking the Root object file that defines the corresponding global variable. Similarly, the constructor without argument is automatically generated by the C++ compiler.

The "get_class_name()" function could easily be generated by the preprocessor. The "get_class_version()" however requires the programmer's cooperation since he sets the version number for the classes. The instantiation of class Meta can also be generated by the pre-processor easily.

The more difficult part for the pre-processor is the generation of serialize() and load(). The pre-processor must parse the class definition, identify the data members and their type and generate the appropriate calls for string conversion (*toS in serialize and Sto* in load). To support every possible user defined type, the pre-processor needs an almost complete C++ parser, which is not trivial given the complex syntax of C++. A simple compromise is to have the preprocessor generate template files from load() and serialize() that the user can fill. Those templates would contain instructions on how to fill them.

4.7 Networking

The network interface between the clients and the servers has been implemented using the Berkeley socket interface [24] [65].

Figure 4.8 illustrates the network interface. On the client side, the loading of the objects is triggered by the smart pointer. All other network functions are called from the base class Object. The networking functions on the client's side prepare the request, open a reliable bidirectional communication channel with the server, send the request and wait for the reply from the server.

The server offers access to its database via a predefined set of functions. Those functions can be accessed by all registered clients. The server has a well known address (socket number) to which all requests are sent. The incoming requests contain the name of the function to access and the arguments for that functions. A dispatcher parses the client's request and calls the appropriate function. Only one request is processed at any given time.

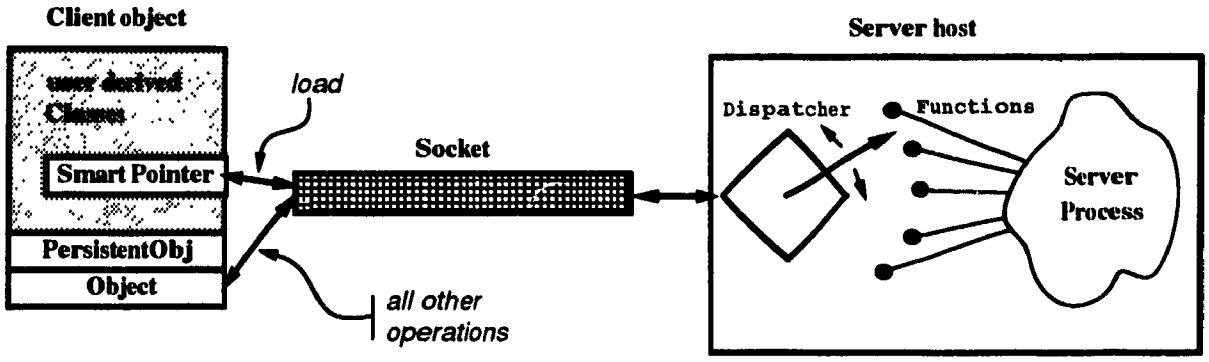


Figure 4.8: Network Mechanism

Other incoming requests wait in a queue. When the function returns with the results, the dispatcher formats a reply packet and sends it back to the client through the connection before taking another request.

This system implies that both the sender and the receiver know exactly what is going to be exchanged, size included. The protocol for the exchange between the client and the server is detailed in the next section.

4.7.1 Object Server External Protocol (OSEP)

OSEP is used to coordinate client/server exchanges. The OSEP is divided in two: communication from client to server, and from server to client.

Communication CLIENT -> SERVER

```
+-----------------------------------------------------
| OSEP version (int) | Function # (int) |  Function Data ...
+-----------------------------------------------------
```

Every message from client to server adopts the convention illustrated above. The first integer represents the protocol version. The protocol version determines how the rest of

the communication is interpreted. A server could be updated to a newer version of OSEP but would still handle calls from older clients using an outdated version of the protocol.

The second field represents the function requested. The function number is unique among all functions offered by the server. The function numbers and names are identified with

```
#define  data_FunctName_V?  n
```

where ? is the version number and n is a number unique identifier for the function/version combination.

This allows the server to provide many versions of the same function simultaneously. With this facility, experimental services can be tested while maintaining regular services.

The first two fields (version and Function number) are called the OSEP header.

The "Data" section of the message is only known by the specific pair of functions (and version) that send (on the client) and receive (on the server) this field. The reply from the server contains the data requested by the client and its format depends on the function called by the client.

Communication SERVER -> CLIENT

```
+---------------
| Function Data ...
+---------------
```

Data Exchange

The Function Data field found in messages sent by clients uses one of three possible formats: *Msg_wrapper*, *Data_wrapper* or *msg_code*. The first is generally used by to communicate commands to the server. It is a structure containing an object name, a client identifier, and a code describing the action requested.

The Data_wrapper is used to exchange persistent objects. It is similar to the Msg_wrapper structure, with an added field for the data string. It is used by commands like: update, save (message to the server) and load (the reply from the server).

The third format named msg_code is simply a number representing an error code returned by the server.

4.8 Notification

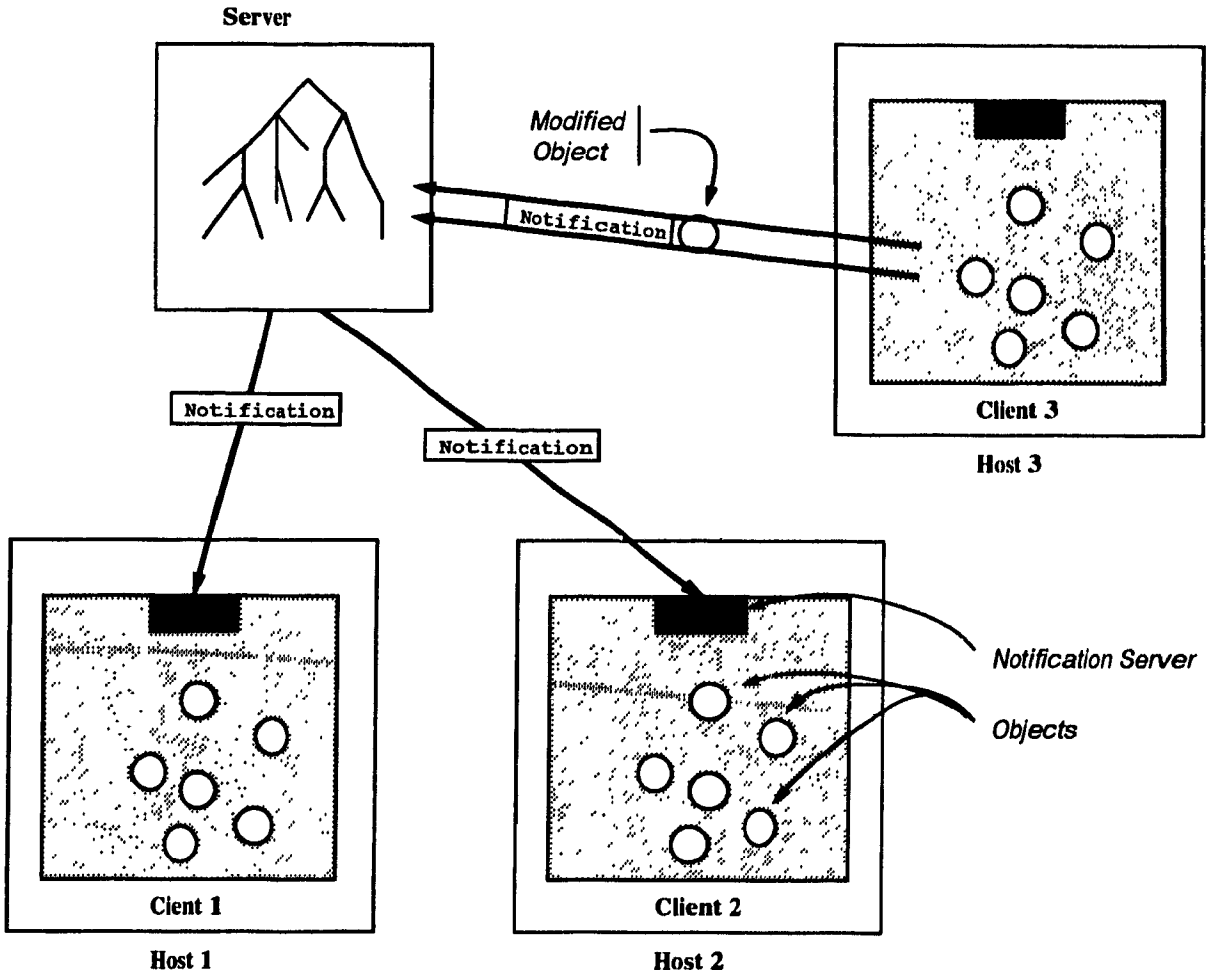


Figure 4.9: Notification Mechanism

The notification mechanism is illustrated in figure 4.9. When notification of data change is issued from the *writing* client, it sends its notification to the server along with a copy of the modified object. The server keeps the modified object in the memory tree without writing it to disk. It then sends messages to the clients registered for notification of this object. The clients are then free to request a copy of the modified object from the server.

Figure 4.9 shows that each client has an embedded notification server. In this figure, Client 3 has the write privilege on the objects while Clients 1 and 2 are registered for notification on those same objects. The notification server receives notifications (both data change and method request) from the object server(s) and processes them. It is the notification server that decides to load the modified copy of the object based on parameters set by the tool it serves. Modified objects can be fetched asynchronously or synchronously.

The notifications server has dynamically defined socket address (i.e. it is a socket attributed by the system at run time). This address is communicated to the object server when a notification request or method export is made. The same socket address is used by all servers interacting with the client. Having a dynamically defined address paves the way for supporting multiple clients on a single host where no predefined well known socket address could serve all clients on a given host.

Chapter 5

Evaluation of the prototype

5.1 Evaluation criterion

This chapter presents an evaluation of the working prototype. The criterion for evaluation are:

- size and complexity of the prototype,
- memory usage overhead,
- disk usage overhead,
- swizzled smart pointer usage performance comparison,
- unswizzled smart pointer usage performance comparison (loading objects),
- integration effort in a user's program,
- comparison of design goals and prototype functionality.

Finally, possibilities for future development are discussed.

5.2 Description of the prototype

The prototype was written in C++ using the gnu C/C++ compiler gcc version 2.4.5. The hardware platform was the sparc workstation (model I+ and II) running the SunOS 4.1.1b operating system.

The implementation of the prototype is a proof of concept for the ideas developed in the architecture described in this thesis. High level classes from the gnu libg++ library are used to simplify the development of the prototype resulting in some costs to performance. It did however, allow efforts to be concentrated on the implementation of the more complicated aspects of the prototype.

The gnu classes used are:

- String: A highly versatile, if somewhat heavy, string class.
- DLList: A double linked list container class.
- AVLMap: An AVL tree based map (used as an associative array) container class.

The code is divided in five subsystems:

- socket lib: A basic socket library written in "C".
- OO net interface: Object Oriented network interface that can sit on top of the socket library or an RPC library. This library also provides the "main" function of the server process which is where the dispatching of incoming messages is handled. The communication protocol is also defined here.
- Server lib: A mini server library to emulate the real backend developed in Nathalie Farjallah's thesis [23].
- main lib: Main library that includes all the base objects for the loaded objects tree and smart pointer classes.
- libg++ classes: Some regular and parameterized classes from the libg++ library [68].

SECTION	blank	comment	source	total
socket lib:	336	381	427	1031
OO net i/f:	585	942	671	2001
Server lib:	214	266	455	828
main lib:	1457	1870	1500	4467
libg++ classes:	719	359	3489	4519

Table 5.1: lines of code per subsystem

The above table shows a breakdown of the amount of code in each subsystem.

From these numbers the coding effort saved by using some classes from the libg++ library to build the prototype can clearly be seen.

5.3 Memory usage overhead

The memory overhead will vary with the length of the pathnames given to each object. Consider an average pathname length of 25 characters which is reasonable for this estimate. Also assume that the objects are under 5 directories that look like "/myproject/spiceView/part-X" where X varies from 1 to 5. Under these directories, each cell is named Cxxx where xxx is a number. The mount point is "/myproject" which mounts "/usr/libraries/scoob" on a server named "bigSur". There are 1000 objects loaded and each is referenced by one smart pointer.

Each object in the loaded object tree inherits the *Object* class. An object can either be an instance of *Dir*, *Root* or *PersistentObj*. In the context of memory usage, the Root class can be considered to be like a Dir. The Dirs are nodes of the tree, while the PersistentObj are leaves. Refer to sections 3.4 and 3.5 for more details.

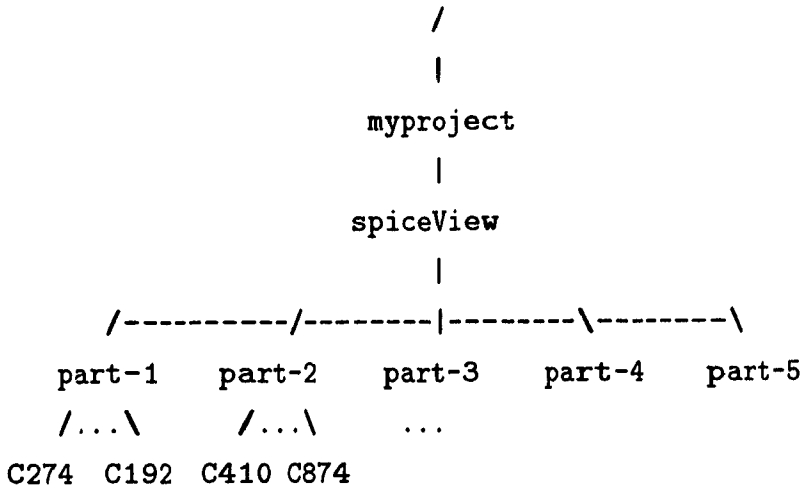
The memory usage for each class is:

- Object: 28 bytes + length of pathname token.

- Dir (regular): 40 bytes.
- Dir (mount point): 40 bytes + string length of server name + mount dir
- Root: 0 bytes.
- PersistentObj: 12 bytes.
- Smart Pointer: 12 bytes + length of full pathname of object referenced.

Note that a Dir can either be a regular dir or a mount point.

The loaded object tree will look like this:



The memory requirement for the tree will be:

- "/": 41 bytes (40 + 1 for the name)
- myproject: 75 bytes (40 + 9 for the name + 6 for bigSur and 20 for /usr/libraries/scoob)
- spiceView: 49 bytes (40 + 9 for the name)
- part-1: 46 bytes (40 + 6 for the name) [5 x 46 = 230]

- C274: 16 bytes (12 + 4 for the name) [1000 x 16 = 16000]

```
Memory tree total:     2225 bytes.
Smart Pointers total: 42 bytes (12 + 30 for /myproject/spiceView/part-1/C274)
      [1000 x 442 = 42000 bytes]
-------------------------------------------------
Grand total:           442225 bytes (43 Kbytes)
```

Looking at the memory overhead for different sizes of objects:

$$overhead = 100 * \frac{overheadfornumOfElements}{objectsize * numOfElements}$$

where numOfElements is 1000.

- 512 bytes objects: 8.6%
- 2K bytes objects: 2.15%
- 8K bytes objects: 0.54%
- 32K bytes objects: 0.27%

The cost is small and could be further reduced by using shorter paths.

5.4 Disk usage overhead

The disk overhead is less critical than the memory overhead since disk space is much cheaper. The overhead in storing the objects in an ascii form can be divided in two parts.

The first is the overhead of encoding the class names, their version and the encapsulation format used to represent strings of ascii data. This overhead for a class "myclass" that inherits from class PersistentObj is 60 bytes.

The second is caused by the fact that an ascii representation of a number usually takes more space than a binary. For instance an unsigned int on a 32 bit computer architecture uses 4 bytes in binary format and can potentially use up to 10 characters, if the number is very large. Here is a breakdown of the overhead for each type of data:

- string: 3-5 char (length, white space and @ sign).
- char: 1 (a char is proceeded by an @ sign)
- long: 1-10 char vs. 4 bytes in binary mode
- short: 1-5 char vs. 2 bytes in binary mode
- uint: 1-10 char vs. 4 bytes in binary mode
- int: 1-10 char vs. 4 bytes in binary mode
- float: 1-10 char vs. 4 bytes in binary mode
- double: 1-20 char vs. 8 bytes in binary mode

If the numeric values stored are under 9999 there is no disk space overhead in using an ascii representation of the data. However, saving and loading could be made faster by using a binary representation of the data, since the binary <-> ascii translation is expensive in terms of CPU cycles.

The reason for using ascii in the first place was to facilitate debugging. Secondly, an ascii representation is portable across different architectures regardless of their byte order (big endian vs. little endian). There exists an eXternal Data Representation library (XDR) distributed with the SunOS operating system (and also available for other platforms) which translates binary data in a network neutral format. Use of this library would preserve the architecture independence, while reaping the performance benefits of a binary format.

While the ascii representation used in the prototype is potentially more space consuming it inflates the data size by a factor of 40% in the very worst case (all fields contain a 10 digits numbers). Considering the fact that large disks are getting cheaper every day, the resulting ration is acceptable even in the worst case, and is good on the average.

5.5 Swizzled smart pointer performance

To make a performance comparison between regular pointers, and swizzled smart pointers a singly linked list was built, where each node is an instance of class *tstclass* defined below.

```
class tstclass
{
  tstclassSmartPtr next;      // smart pointer to next element in list
  String           somedata;  // String representing the data
}
```

The test program cycles 100,000 times through a list of 1000 elements.

5.5.1 List traversal with Smart Pointers

Given below is the code of the test with some contextual information. The code is commented with a rough approximation of the number of machine instructions that each line will generate.

```
tstclass *curObj;          // normal ptr used as cursor during list traversal
tstclass *nextObj;         // normal ptr used as cursor during list traversal
tstclassSmartPtr listHead; // Initialized to head of the list.
```

The operators "=" and "->" have been overloaded to allow the smart pointer to provide automatic translation to a regular pointer on an assignment and automatic swizzling of the object if needed when dereferencing the smart pointer.

```
<T>*  operator->() {    // If unswizzled: swizzle smart pointer
  if (obj_ptr != NULL)
    return (obj_ptr);
```

```
    else
      return (swizzle());
  }

  operator <T>* () {      // allows assignment to regular pointer
    if (obj_ptr != NULL)
      return (obj_ptr);
    else
      return (swizzle());
  }
```

Both operators are inline. If the smart pointer is swizzled, on a risc architecture machine the approximate cost (in machine instructions) of operator "=" is 2 instructions and operator "->" is also 2 instructions.

Test code annotated with number of machine instructions per line:

```
cycleCount = 0;
while (cycleCount < numOfCycles)            // 5
  {
    curObj = listHead;                      // 2 + 2 = 4  <<<< overload
    elementCount = 1;                       // 2
    while (elementCount != numOfElements)   // 5
      {
        nextObj = curObj->next;             // 3 + 2 = 5  <<<< overload
        curObj = nextObj;                   // 2
        elementCount++;                     // 6 for ++ & jump
      }
    cycleCount++;                           // 6 for ++ & jump
  }
```

```
Total:  cycleCount * ([outer loop]  + (numOfElements * [inner loop]))
Total:  100,000    * (17 + (1000 * 18)      = 1.802 x 10E9 instructions
Time to run the test:  158 seconds
```

5.5.2 List traversal with regular pointers

For the regular pointer test the class was the same except for the replacement of the smart pointers with regular pointers.

```
cycleCount = 0;
while (cycleCount < numOfCycles)              // 5
  {
    curObj = listHead;                        // 2    <<<<<  different
    elementCount = 1;                         // 2
    while (elementCount != numOfElements)     // 5
      {
        nextObj = curObj->next;               // 3    <<<<<  different
        curObj = nextObj;                     // 2
        elementCount++;                       // 6 for ++ & jump
      }
    cycleCount++;                             // 6 for ++ & jump
  }
```

```
Total: cycleCount * ([outer loop] + (numOfElements * [inner loop]))
Total: 100,000    * (15 + (1000 * 16)      = 1.601 x 10E9 instructions
Time to run the test:  81 seconds
```

5.5.3 Analysis of results

The ratio of machine instructions for regular pointers / smart pointers is

$$100 * \frac{1.601 \times 10E9}{1.802 \times 10E9} = 89\%$$

while the time ratio is

$$100 * \frac{81sec}{158sec} = 51\%$$

.

It is safe to assume that the difference in execution time can be attributed to the two lines of code identified as different in the test code. This time difference is 158 - 81 = 77 seconds. After verification with the assembler code generated by the compiler, it was found that the compiler did not inline the overloaded functions because the debug (-g) switch was selected for the compile instead of the optimize (-O). Inlining the code would make the time ratio similar to the instruction ratio which would mean 89% of the efficiency of regular pointers (expected time of 91 sec).

The other operations common to both versions of the test can be attributed for the remaining 81 seconds, The machine instructions for those common lines is estimated to be around 13 for the outer loop and 13 for the inner loop.

This is the minimum code needed to traverse a singly linked list without doing anything else. To do anything meaningful to each node during the traversal the time per iteration in the loop could easily be augmented by a factor of 2 to 4 (as seen in the test a single function call almost doubled the time), and thereby bringing the list traversal speed of smart pointer to 94-97% of the regular pointer. In practice however, one could expect much heavier operations to be executed in the loop, which could help to further minimize the difference between a swizzled smart pointer and a regular pointer.

5.6 Object loading and saving performance

Next, is a comparison of the loading and saving times of objects using the smart pointer library versus regular NFS based disk access.

5.6.1 Test description

To perform all the development and tests a substitute server was used, since the real server [23] was not completely functional. The substitute server has a subset of the functionality of the real server that is sufficient to evaluate the prototype.

The substitute server

The substitute server is made of two files, an index file and a data file. The data file stores the ascii string that represents the objects. Each entry has a fixed size which is slightly larger than the average object size. The index file contains a list of object names and their offset in the data file. When starting the server, the index file is loaded into a memory AVLMap, that is used as an associative array to get the offset of a given object name. This solution is very simple (given the use of the AVLMap from libg++) yet reasonably efficient.

The front end of the server was already written and ready to interface with the real server so no extra work was required.

Description of the NFS disk load/save test

Due to the different methods used for the reading and writing information between the two systems, a crude and straight forward implementation of direct disk reading and writing of objects was made.

The test borrows the disk access primitives from the substitute server to read from the index file, and retrieve the ascii string encoded object. One by one, it loads the objects from

file and dynamically allocates the memory in an attempt to emulate the basic behavior of a working system. The system timestamp (granularity: 1 second) is printed once at the beginning of test test and again when the loading is finished.

The second part of the test opens a new file in the same directory as the original one and writes the objects to it one at at time. When this is finished, the system time is printed again to get the write time.

Because this test code shares the same disk data format as the substitute server, it is natural to use the data files generated by the testing of the persistent object saving scheme.

No attempt was made at trying to make the object storage non sequential in the disk, mainly because it would have involved rewriting another set of disk access routine. The down side to this is that the disk reading/writing takes full advantage of UNIX/NFS disk caching thus considerably decreasing the read and write times for smaller objects.

Description of the Persistent object load/save test

This part of the test is divided in three programs, the server, the writing test and the loading test.

First the writing test is run, which creates a singly linked list made of a given number of persistent objects of a specific size. Each object points to the next using a smart pointer. Once created it proceeds to print the system time, traverses the list, saving each element to the server, and prints the system time once finished.

The load test initializes a smart pointer with a well known name (the name of the list head) and starts traversing the list. Each time it tries to dereference a smart pointer through "->" the next object is loaded. The system time is printed at the beginning and the end of the load test.

Note that no optimization switches were enabled, to guarantee that the compiler would not optimize out the test. Despite this precaution, it was observed earlier that the compiler did not inline some operator overload as expected and therefore the smart pointer tests showed slower results than expected.

Test setup

The tests were executed using the following setup:

- client machine: sparc-II, 52Meg RAM, 90Meg swap space,
- server machine: sparc-I+ 28Meg RAM, 130Meg swap space,
- operating System: SunOS 4.1.1b,
- compiler: gcc 2.4.5 with -g option,
- network: ethernet with collision rates peaking at 15%. Care was taken to run the tests off hours and monitored for excessive collision rates. Empirical measures showed that an error margin of up to 10% could be caused by network collisions.

5.6.2 Results

The tests were made using objects of size ranging from 512 bytes to 32K. The number of objects ranged from 10 to 1000.

OBJ SIZE	512	2K	8K	32K
10 objects:	3	2	2	5
100 objects:	21	21	21	48
1000 objects:	202	200	206	420

Table 5.2: load time using smart pointers (in seconds)

OBJ SIZE	512	2K	8K	32K
10 objects:	1	1	1	2
100 objects:	1	1	4	21
1000 objects:	2	14	46	205

Table 5.3: load time using nfs (in seconds)

OBJ SIZE	512	2K	8K	32K
10 objects:	4	3	3	5
100 objects:	21	20	21	44
1000 objects:	233	201	211	480

Table 5.4: write time using smart pointers (in seconds)

OBJ SIZE	512	2K	8K	32K
10 objects:	1	1	1	3
100 objects:	1	1	7	26
1000 objects:	3	16	65	366

Table 5.5: write time using nfs (in seconds)

5.6.3 Results analysis

As expected, reading/writing on the server server end and transmitting objects between the server and client through sockets requires significantly more time than having each client directly reading from disk. However, going through a server makes a number of advanced services possible.

Times for load and save of persistent objects ranging in size from 512 to 8K are all identical. This suggests that there is a fixed overhead associated with the loading and creation of new objects which greatly outweighs the manipulation time of the data for small objects (smaller or equal to 8K). Loading objects four times larger (32K) only takes double the time. This shows an area for performance improvement: the base overhead for small objects. This improvement is a fixed overhead not proportional to the amount of data loaded or saved since the times are identical for objects from 512 to 8K. When the objects are loaded, a memory allocation is made and the constructors are called to create the new objects. When saving however, the objects are not freed and no constructors or destructors are called. The only operations in common are the socket communications to the server. In other words, to improve performances on small objects, efforts have to be concentrated on network link and server overhead reduction.

The 8K boundary coincides with the 8K page size used by NFS/UDP and might have some effects on the performance jump between 8 and 32K. However, it cannot be the sole explanation for the differences since both the persistent object and the NFS scheme show these differences (although in different proportions).

This jump in the times can be attributed to some of the inefficiencies of the prototype (the implementation, not the concepts). In the current prototype, the data is copied many times from one layer to the next, before it is finally copied to the right data member of the object (an instance of class String).

In the NFS based test, the times for small objects (<= 8K) in quantities of 10 and 100 are around 1 second. These times compared with the same objects in quantities of 1000 or larger objects (32K) clearly show the effect of the NFS cache on small objects (a larger number of small objects can fit in the cache). This explains the large difference in times

between the smart pointer and the NFS approach for those small objets.

Any useful comparison between the smart pointer and the NFS scheme is made impossible for those small objects because the NFS cache reads them all in one operation. It would require too great a number of small objects to make the cache effect negligible.

A major optimization for large objects would be to design a more specialized string class that could accept a pre-allocated buffer containing a string as an argument to one of its constructors, rather than allocating a new buffer and recopying the string.

A detailed profiling exercise would be needed to precisely pin point other major sources of improvements. The name lookup in the loaded object tree is expected to be another weak point since none of these operations have been optimized for speed in the prototype.

No matter what improvement are made, the results will always be slower than NFS access. This is the cost of having access to a database. Optimizations can however, try to minimize this cost.

Another study [23] shows a clear comparison of read/write times of various access types for VLSI objects. The performance comparisons are similar to the ones found here (for the larger objects), and clearly show that the approach taken in this work is nevertheless faster than interfacing to a relational database.

5.7 Comparison of design goals and prototype functionality

This section looks back at the original design goals and looks at the prototype to measure how well these goals were achieved:

- support for complex object: Full support
- version control: Full support of version number and timestamps.
- inter process notification: Not supported by the prototype.

- automatic version translation: Supported.
- ease of development for new tools: The detailed steps needed to use the library are described in section 4.6. The pre-processor has not yet been written. However, template files exist that the user can fill for the load and serialize function and a single quick-reference sheet should be enough for a user to feel comfortable with the library.
- ease of integration of existing tools: If the user program is single threaded and written in C or C++, the user can simply modify his classes to inherit from the PersistentObj class and follow the other steps to get things working. Note that change a struct into a class, all you have to do is to change the word "struct" to "class" and add the keyword "public:" on the first line of the definition. The program can keep using regular pointers to access the objects because the program is single threaded and therefore no other process can unload objects without the main program's knowledge. The initial load is done by dereferencing a smart pointer containing the name of the object to load (the same pointer can be used many times by just changing it's name). The unload and save are taken care of automatically at the program termination.
- portability: This system is directly portable to any UNIX system that supports the BSD sockets, and for which there is a port of the gnu C++ compiler (it currently supports 33 architectures).
- versatility in client/server distribution: The prototype is fully versatile in its ability to mount any number of servers on the loaded object tree for transparent access by the application.

5.8 Future development

Future development can be divided in three parts, features planned and designed for release 1.0 that did not make it into the prototype, performance improvements and new functionality.

Features that did not make it in the prototype:

- dynamic type checking,
- pre-processor to help automate library interface stub generation,
- data and function notification,

Areas of performance improvements:

- minimize the recopying of data through the different layers,
- use XDR protocol and binary disk format to save space and run faster,
- use a UDP with packet count and retransmission protocol similar to NFS rather than the heavier TCP/IP,
- do detailed profiling to identify further improvements.

New functionality:

- a high level project manager,
- a sophisticated version and configuration management,
- study the possibilities for integration with CORBA communication infrastructure.

5.9 Conclusion of the evaluation

This chapter evaluated the prototype CAD storage framework. The results show that most of the functionality described in the previous chapter was successfully implemented in the prototype. The memory and disk overhead were shown to be acceptable and in most applications the performance of a swizzled smart pointer is comparable to that of a regular pointer.

The loading and saving of objects with the prototype was observed to be about twice as slow as loading the same object directly from NFS (for large objects). The overhead for small objects is higher and useful comparison with NFS is made impossible because of the NFS caching. Major areas and methods for improving performances were identified and areas of interest for future development were listed.

Chapter 6

Conclusion

The goal of this thesis is to satisfy the need for a modern CAD VLSI framework. To meet this requirement a study of some existing systems, their strengths and shortcoming is presented. A list of desirable features for a modern CAD framework is laid out and the architecture of the foundations for such a framework is described.

The architecture describes a multi client / multi server scheme where access to the server(s) is transparent to the client application. The server part is implemented in another thesis [23]. It is an object oriented database management system (OODBMS) with versatile storage granularity, a no-overwrite policy, concurrency control and basic versioning.

This thesis concentrates on the client library and networking facilities but also describes the concepts of the global architecture. The architecture describes the concepts and implementation methods to provide rich modeling capabilities, version control, concurrency control and transparent network access to multiple databases by a single client through object persistence.

The implementation is done with an object oriented language allows easy integration of existing tools and easy migration for developers using the "C" language. The language chosen is "C++".

The heart of the client architecture is a base class library from which the user's classes are derived to obtain object persistence and other features listed above.

In addition to these basic requirements, the thesis describes automatic version translation upon loading an object instantiated from an older version of the user's classes. It is also concerned with portability across multiple UNIX platforms allowing clients of different architectures to access the same server. Portability is achieved through the use of "C++" and the Berkeley sockets library available on most UNIX platforms.

A description of more advanced features such as project manager, data and method notification is also presented.

As proof of concept, a working prototype of the client library and networking modules has been developed and tested. The prototype implements all of the concepts discussed above with the exception of notification and project manager.

Although the prototype is not implemented with performance or resource saving in mind, the test results show some moderate costs in memory, disk and processing overhead in the object persistence model.

Areas of performance improvements and methods to achieve them are described and ideas for future research and development are presented; the main topics are notification, project management and integration of CORBA compliant communication infrastructures.

The architecture is functional and with some performance improvements could be used as the foundation for a working highly versatile and expandable CAD framework.

Bibliography

[1] M.P. Atkinson, P. Buneman, and R. Morrison. *Data Types and Persistence.* Springer-Verlag, 1988.

[2] M.J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986.

[3] S. Banks, C. Bunting, R. Edwards, L. Fleming, and P. Hackett. A Configuration Management System in a Data Management Framework. In *28th ACM/IEEE Design Automation Conference*, pages 699–703, 1991.

[4] S. Benrabah. Vérification interactive de circuits par l'analyse incrémentale. Master's thesis, École Polytechnique de Montréal, 1990.

[5] B. Berliner. CVS II: Parallelizing Software Development. In *USENIX winter conference*, 1990.

[6] A. Biliris. Database Support for Evolving Design Objects. In *26th ACM/IEEE Design Automation Conference*, pages 258–263, 1989.

[7] P. Bingley and P. van der Wolf. A Design Platform for the NELSIS CAD Framework. In *27th ACM/IEEE Design Automation Conference*, pages 146–149, 1990.

[8] G. Booch. *Object Oriented Design with applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

[9] G. Booch and M. Vilot. Object persistence. *C++ Report*, pages 12–16, Mars-April 1993.

[10] M.A. Breuer, W. Cheng, R. Gupta, I. Hardonag, E. Horowitz, and S.Y. Lin. Cbase 1.0: A CAD Database for VLSI Circuits Using Object Oriented Technology. In *(ICCAD) International Conference on Computer-Aided Design*, pages 392–395, 1988.

[11] Cadence Design Systems, Inc. *SKILL Language Reference Manual Version 2.1*, June 1989. Edge Distribution.

[12] G-D. Chen and T-M. Parng. A database mmanagement system for a VLSI design system. In *25th ACM/IEEE Design Automation Conference*, pages 257–262, 1988.

[13] H-T. Chou and W. Kim. Version and Change Notification in an Object-Oriented Database System. In *25th ACM/IEEE Design Automation Conference*, pages 275–281, 1988.

[14] D. Comer. *Internetworking with TCP/IP. Principles, Protocols and Architecture.* Prentice Hall, 1988.

[15] J.R. Corbin. *The Art of Distributed Applications.* Springer-Verlag, 1990.

[16] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[17] D.H. Craft. A study of pickling. *Journal of Object-Oriented Programming*, 5(8):54–66, jan 1993.

[18] J. Daniels and S. Cook. Strategies for sharing objects in distributed systems. *Journal of Object-Oriented Programming*, 5(8):27–36, jan 1993.

[19] O. Deux and al. The Story of O2. *Transactions on Knowledge and Data Engineering*, 2(1):91–108, Mars 1990.

[20] K.R. Dittrich and R.A. Lorie. Version Support for Engineering Database Systems. *Transactions on Software Engineering*, 14(4):429–436, April 1988.

[21] D.J. Ecklund and F. Tonge. A context mechanism to control sharing in a design database. In *25th ACM/IEEE Design Automation Conference*, pages 344–350, 1988.

[22] M.A. Ellis and B. Stroustrup. *The annnotated C++ reference manual.* Addison Wesley, 1990.

[23] N. Farjallah. Un Système de Gestion de Base de Données par Objets. Master's thesis, École Polytechnique de Montréal, Département de Génie Électrique, To be published.

[24] J. Frost. BSD Sockets: A Quick And Dirty Primer. This Socket Tutorial is available at anonymous ftp sites on the Internet or contact jimf@saber.com, Aug 1990.

[25] D. Gedye and R. Katz. Browsing the Chip Design Database. In *25th ACM/IEEE Design Automation Conference*, pages 269–274, 1988.

[26] D.S. Harrison, P. Moore, R.L. Spickelmier, and A.R. Newton. Data Management and Graphics Editing in the Berkeley Design Environment. In *(ICCAD) International Conference on Computer-Aided Design*, pages 24–27, 1986.

[27] K. Haviland and B. Salama. *UNIX System Programming.* Addison-Wesley, 1987.

[28] S. Heiler, U. Dayal, J. Orenstein, and S. Radke-Sproull. An Object-Oriented Approach to Data Management: Why Design Databases need it. In *24th ACM/IEEE Design Automation Conference*, pages 335–340, 1987.

[29] C. Horn. Standardizing your object interface. *Object Magazine*, pages 56–64, Sept-Oct 1993.

[30] A. Hsu and L-H. Hsu. HILDA: An Integrated System Design Environment. In *IEEE International Conference on Computer Design*, pages 398-402, 1987.

[31] M.L. Katz, T.J. Mowbray, R. Zahavi, and D.L. Cornwell. System Integration with Minimal Object Wrappers. In *TOOLS USA 93*, pages 31–42, 1993.

[32] R.H. Katz, M. Anwarrudin, and E. Chang. A version server for computer-aided design data. In *23th Design Automation Conference*, pages 27–33, 1986.

[33] B.W. Kernighan and D.M. Ritchie. *The C programming language.* Prentice-Hall, 1978.

[34] W. Kim. Object-Oriented Databases: Definition and Research Directions. *Transactions on Knowledge and Data Engineering*, 2(1):327–341, Sept. 1990.

[35] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, Mars 1990.

[36] D.E. Knuth. *The art of computer programming*, volume 1. Addison-Wesley, second edition, 1973.

[37] J. Koivisto and J. Reilly. Cerial: A tool for XDR or ASN.1 BER serialization of C++ Objects. In *TOOLS USA 93*, pages 109–122, 1993.

[38] P. Laurent and N. Silverio. Persistence in C++. *Journal of Object-Oriented Programming*, 6(6):41–46, Oct 1993.

[39] W. Li and H.Switzer. A Unified Data Exchange Environment Based on EDIF. In *26th ACM/IEEE Design Automation Conference*, pages 803–806, 1989.

[40] S.B. Lippman. *C++ Primer, 2nd edition.* Addison-Wesley, 1991.

[41] L-C. Liu, P-C. Wu, and C-H. Wu. Design Data Management in a CAD Framework Environment. In *27th ACM/IEEE Design Automation Conference*, pages 156–161, 1990.

[42] B. Meyer. *Eiffel - The Language.* Prentice-Hall, 1992.

[43] T.J. Mowbray and T. Brando. Interoperability and CORBA-based open systems. *Object Magazine*, pages 50–54, Sept-Oct 1993.

[44] G. Nelson. *Systems Programming with Modula-3.* Prentice-Hall, 1991.

[45] Object Mamagement Group, Framingham (MA). *The Common Object Request Broker: Architecture and specification, Document 91,12,1*, Dec 1991.

[46] I. Pohl. *C++ for C programmers.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[47] S. Prata. *The Waite Group: Advanced UNIX - A Programmer's Guide.* Howard W. Sams & Company, 1989.

[48] R. Probst. A software model for the 21st Century. Distributed computing is everywhere. *Object Magazine*, pages 65–67, Sept-Oct 1993.

[49] J.E. Richardson and M.J. Carey. Persistence in the E Language: Issues and Implementation. *Software - Practice and Experience*, 19(12):1115–1150, Dec. 1989.

[50] M.J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, Dec. 1975.

[51] S.M. Rubin. *Computer aids for VLSI design.* Addison-Wesley, 1987.

[52] S.M. Rubin. A General-Purpose Framework for CAD Algorithms. *IEEE Communications Magazine*, pages 56–62, May 1991.

[53] M. Silva and R. Newton D. Gedye, R. Katz. Protection and Versioning for OCT. In *26th ACM/IEEE Design Automation Conference*, pages 264–269, 1989.

[54] R.L. Spickelmier. A User's Guide to Oct 2.0. Technical Report EPM/RT-88-36, University of California at Berkeley, July 1989.

[55] R.L. Spickelmier, P. Moore, and A.R. Newton. A Programmers's Guide to Oct. Technical report, University of California at Berkeley, March 1990.

[56] R.L. Spickelmier and A.R. Newton. A User's and Programmers's Guide to RPC: Remote Procedure Call Package for Oct/VEM. Technical report, University of California at Berkeley, Feb. 1990.

[57] G.L. Steele. *Common Lisp: The Language, 2nd Edition.* Digital Press, 1990.

[58] W.R. Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[59] M. Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Second International Conference on Data Engineering*, Los Angeles (CA), Feb. 1986.

[60] M. Stonebraker. The Design of the POSTGRES Storage System. In *VLDB Conference*, Brighton, England, 1987.

[61] M. Stonebraker. Future trends in database systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):125–142, Mars 1989.

[62] M. Stonebraker and L. Row. The Design of POSTGRES. In *ACM-SIGMOD Conference*, Washington, D.C., June 1986.

[63] M. Stonebraker, L. Rowe, and M. Hirohama. Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, Mars 1990.

[64] B. Stroustrup. *The C++ Programming Language, second edition.* Addison Wesley, 1991.

[65] Sun Microsystems, Mountain View (Calif.). *Network Programming Guide*, Mars 1990. SunOS 4.1.1 Distribution.

[66] A.S. Tanenbaum. *Computer Networks, 2nd Edition.* Prentice-Hall, 1988.

[67] W.F. Tichy. Design, Implementation and Evaluation of a Revision Control system. In *IEEE 6th International Conference on Software Engineering*, 1982.

[68] M. Tiemann and R.M. Stallman. *libg++, GNU Project.* Free Software Foundation Inc., 1991.

[69] P. van der Wolf and T.G.R van Leuken. Object Type Oriented Data Modeling for VLSI Data Management. In *25th ACM/IEEE Design Automation Conference*, pages 351–356, 1988.

[70] S. Vinoski. Distributed object computing with CORBA. *C++ Report*, pages 34–38, July-Aug 1993.

[71] I. Widya, T.G.R. van Leuken, and P. van der Wolf. Concurrency control in a VLSI design database. In *25th ACM/IEEE Design Automation Conference*, pages 357–362, 1988.

[72] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *Transactions on Knowledge and Data Engineering*, 2(1):63-75, Mars 1990.

[73] M. Winslett, D. Knapp, K. Hall, and G. Wiederhold. Use of change coordination in an information-rich design environment. In *26th ACM/IEEE Design Automation Conference*, pages 252-257, 1989.

[74] M.S. Yoo and A. Hsu. DEBBIE: A Configurable User Interface for CAD Frameworks. In *(ICCD) International Conference on Computer Design*, pages 135-140, 1990.