I dedicate this work to my wife Josephine and my parents Ruth and Donald.
Tool and Database Interfacing

based on Virtual Objects

Proefschrift
ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus,
prof. drs. P. A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
aangewezen door het College van Dekanen
op maandag 11 januari 1993 te 14.00 uur.

doors

Matthew Neil Sim

geboren te London
elektrotechnisch ingenieur
Dit proefschrift is goedgekeurd door de promotie prof. dr. ir. P. M. Dewilde.

Published and distributed by:

University Printers T.U. Delft
Kanaalweg 2b
The Netherlands
Summary

Software systems that perform many tasks have a complex structure. Developers of software systems manage the complexity by constructing the system from several parts. Each part, called a software module, performs a specific task. A tool is a type of software module that enables the user of a system to perform a task. The number of tools in the system increases with the diversity of tasks the user performs. Data is common if it is created in one tool and used in other tools. Data is persistent if it exists after the program (the tool) that created the data has stopped running. Many systems use a database module to manage common and persistent data. The task of the database is to coordinate the use of the common data among the tools and to pass the persistent data to a storage repository. The modular structure of the system requires the modules to communicate with each other. A programming interface is a special module that enables the tool programmer to control communication. At the time this research began, the common programming interface was procedural. A procedural programming interface is a set of procedures and data structures.

The complications of procedural interfacing are:
(i) The tool programmer must learn the set of procedures and data structures.
(ii) The programmer must insert the procedures into the tool program in order to control the transport of data between the tool and the database.
(iii) The structure of the data manipulated by the tool often differs from the data structure used by the database.

These complications motivated researchers to develop a programming interface that is simpler for the tool programmer to use. The approach we have taken is to make objects appear virtual to the tool programmer. A virtual object exists in the memory space of an executing tool whenever it is needed. If the interface supports virtual objects then the programmer does not need to program the transport of data between tools and the database so there is no need to insert interfacing procedures.

We use the term Virtual Object Interface System (VOIS) to describe an interface that supports virtual objects. In this thesis we discuss the concepts of virtual object interfacing, address several problems and provide solutions based on an experimental VOIS, called OPI. We investigate the utilization of the system to enhance data management and inter-tool communication. This study focuses on the interfacing of electronic design tools used for the design of compact electronic circuits.

OPI is realized in an object-oriented programming language. In such a language the programmer views data in the program as a group of interacting objects. The objects send each other requests to perform various tasks. Inheritance and encapsulation are important
properties, supported by the object-oriented language, for implementing the virtual objects. Inheritance enables one object type to inherit data structure and behavior from another object type. In OPI, an object type exists that is virtual. Using the inheritance property, object types developed by the tool programmer may inherit virtuality. Encapsulation means that the internal data of an object cannot be directly accessed by other objects; they must send a request to the object. Encapsulation enables objects to exist in the memory space of the executing tool without actually loading in the internal data stored in the object from the database. The data is only loaded into the object when it receives a request.

In chapter 1 we introduce the interfacing problem and the traditional solution in the area of electronic design. We discuss the software modules that participate in the interfacing and how they are stratified. We also present the characteristics of several interface systems. In chapter 2 we discuss how to automate the interfacing process. We analyse how data is transferred and define the operations that need to be automated. We motivate the selection of an object-oriented language to implement the interface system and also present several other interface systems. In chapter 3 we present the management modules necessary to perform the interfacing in OPI. We present an overview of the run-time environment and how OPI fits into a programming environment. Chapter 4 presents the properties of the virtual object and how these properties are implemented. In chapter 5 we discuss how objects that are to be stored together are grouped and how the programmer may control this. We present partitioning algorithms. Preliminary results show that the partitioning time is linear. In chapter 6 we describe how the OPI kernel can be extended to support design management of small objects. In chapter 7 we propose how to use the VOIS kernel to support concurrent data sharing between tools. Chapter 8 presents a test system. The programming environment should have a test system to test tools being developed. The approach we take is to view a test as an object, and use an object-oriented database to manage the tests. In chapter 9 we present a case study of a procedural interface called CFI-DRPI. We provide the reader with an insight into the complexity of conventional database interfacing. In chapter 10 we present a case study on the implementation of the CFI-DRPI interface using OPI. The preliminary measurements indicate that the performance of OPI may be sufficient to offer an alternative to the procedural interface. In chapter 11 we reflect on the work covered in the previous chapters, give several suggestions based on the author's experience and make a projection.

To conclude, the author believes that the dependence on an object-oriented language is acceptable since this language type is also effective for implementing large software systems. Preliminary results indicate that if the VOIS uses a good partitioning algorithm then the performance is acceptable. The VOIS forms a good basis for enhancements.
# CONTENTS

1. Introduction .................................................. 1
   1.1 Chapter Overview ........................................... 1
   1.2 The Interfacing Problem .................................... 1
   1.3 The Conventional Solution - System Stratification ........... 4
   1.4 Requirements ............................................... 6
   1.5 Example: Some Characteristics of Several Tool Interfaces .... 8

2. Virtual Object Interfacing .................................... 11
   2.1 Chapter Overview ............................................ 11
   2.2 Basic Persistency Problems and Solutions ..................... 12
   2.3 The Object Oriented Paradigm ................................ 15
   2.4 A Chronological Perspective ................................ 17
   2.5 Original Persistent Systems ................................ 23

3. OPI - an Experimental VOIS .................................... 27
   3.1 Chapter Overview ............................................ 27
   3.2 OPI Architecture ............................................ 29
   3.3 Virtual Objects in OPI ....................................... 33
   3.4 The Application Programming Environment ..................... 38

4. Object Heap Management ....................................... 41
   4.1 Chapter Overview ............................................ 41
   4.2 Choosing an Information Data Model ......................... 42
   4.3 Virtual Object Management .................................. 43
   4.4 Object Aggregation .......................................... 48
   4.5 Aspects of Dynamic Object Management ....................... 53
   4.6 Virtual Object Addressing ................................... 61
   Appendix 4.1 The Programming Interface ....................... 64
   Appendix 4.2 A Variant Transition Mechanism .................... 69

5. Partitioning Management ....................................... 71
   5.1 Chapter Overview ............................................ 71
   5.2 Refinement of the Problems Concerning Data Configuration ... 72
   5.3 Partitioning Principles ..................................... 72
   5.4 Run-time Data Configuration Enforcement ..................... 80
   Appendix 5.1 Local Universe Variant for Virtual Object Addressing . 85
   Appendix 5.2. Implementation Aspects ......................... 85
   Appendix 5.3 The Configuration Data Schema ..................... 86
   Appendix 5.4 The FEW_UNPARTITIONED Algorithm ................ 90

6. Fine Grain Data Management Utilizing a VOIS .................. 95
   6.1 Chapter Overview ............................................ 95
   6.2 ECAD Environments ......................................... 96
   6.3 Modeling Concepts .......................................... 96
   6.4 Considerations And Problems ................................ 97
   6.5 Modeling of Fine-Grained Design Objects .................... 97
   6.6 Incorporation in a VOIS ..................................... 99
   6.7 Overview of the Architecture ................................ 100
   6.8 Extending the Virtual System ................................ 101
   6.9 A VLSI Interface Case Study ................................ 103
Chapter 1

Introduction

1.1 Chapter Overview

Interfacing an application program (tool) with a database management system (DBMS) gives rise to problems due to the difference in the structure and properties of the data (i.e. volatile data in computer memory on the one hand and persistent data on disk on the other). Therefore, much activity in the computer-aided-design (CAD) system engineering community focuses on simplifying the interfacing process. The goal is to obtain convergence between programming languages and databases to produce a coherent entity. The traditional technique is to use a procedural interface, which is a well defined set of procedures and data-structures that the programmer may utilize. In the field of ECAD, coherent systems are not yet ready to replace database systems with procedural interfaces, due to reasons such as lack of standards, degraded speed, the existing huge investment in conventional systems and an incomplete view of the total problem.

A commonly accepted approach towards this goal is to create an interface system that performs fine-grained data management, to bridge the gap between the volatile medium of tools and the persistent medium of databases. The DBMS should provide an interface for large-grained objects. It is important that the interface system is an independent module to provide an interface to various tools and DBMS systems.

This chapter is structured as follows: Section 1.2 discusses the interfacing problem. Section 1.3 presents the major software levels utilized in a conventional ECAD system. Section 1.4 lists interface requirements. Section 1.5 discusses characteristics of several tool DBMS interface systems. (The material presented here is compiled from [1, 2]).

1.2 The Interfacing Problem

We first provide several definitions.

Definitions

A tool is a software program. Its execution performs a specific task.
A volatile object is an object resident in the working memory.

A persistent object is an object that is stored in a database or if volatile is destined to persist beyond the lifetime of the program that created it.

A transient object is a volatile object that does not persist.

An application object is a volatile or persistent object that was created by an application program.

A database object is an abstraction over data stored in the database that may be associated with a unique database key. This key is called the object's persistent identifier (PID).

Check-out of data is acquiring persistent objects from a DBMS.

Check-in of data is presenting a DBMS with persistent objects for storage.

A package is a coherent collection of data that may be input data to, or output data from, a tool.

Import of data is putting a package into a tool from an external source, such as a database or another tool.

Export of data is out-putting a package from a tool to an external destination.

Transportation is importing or exporting.

A connected object is a persistent object that is an integral part of the component of the volatile objects of a running tool to which it is connected.

Implosion of data is disconnecting a persistent object resident in working memory of a tool and transforming it to a package.

Explosion of data is transforming a persistent object in packaged form to a connected object in the working memory of a tool.

Transformation is imploding or exploding.

A cluster is an aggregation of persistent objects that are imploded as a whole and that together they form a package or a sub-unit of a package.

Loading is importing data from a database and exploding it.

Saving is imploding data and exporting it to a database.

Figure 1 illustrates the difficulties of interfacing a tool with a DBMS. Figure 1.1 depicts the user interface in terms of a set of database functions that the tool programmer may apply. The set may be split into two sub-sets, check-in and check-out. The check-out functions import data into a tool's workspace, and the check-in functions export (modified) data back to the database. It is clear that the tool programmer must explicitly program the database functions into the tool.
code. This creates a tight binding between the tool and the database with the result that substantial recoding of the tool would be necessary to make the tool usable with a different type of database. The binding of a tool to a database requires additional functionality to the database function set, as illustrated in figure 1.1.B. The structure of the data manipulated by the tool is oriented towards working memory manipulation (memory allocated by the host machine for the use of the running tool), making efficient use of pointers. The persistent image cannot support pointers between uncontiguous data, thus additional code must be written to perform the mapping between the persistent and the volatile data formats.

![Diagram of interface problem](image)

**Figure 1.1. The interface problem.**

Tool DBMS interfacing is a recognized problem and various publications discuss the subject. The following quote wraps it up: "A considerable amount of code, typically 30% is concerned in transferring data to and from files or DBMS's. Much space and time is involved in coding the translators, and the quality of software can be impaired by the mapping" [3]. Balch et al [4] point out inherent weaknesses in current file based operating systems and suggest their replacement with a persistent virtual memory and propose a layered series of interfaces for persistent store. We quote Straw [5], "A further goal is to provide database support in such a way that it puts as little burden as possible on the applications programmer. The programmer should have a single syntax for access to objects, be they in memory or on disk. The system should deduce permanent status of objects, with no requirement on the programmer to signal writing an object to disk". A popular view among CAD engineers is that programming languages and databases should converge to a coherent and consistent unity, so that data definition, data manipulation and general computation are combined into one language to avoid "impedance mismatch" [6]. (A term used to describe the inconsistency between data structure and functions
between the volatile and persistent environments of the data).

Currently, many organizations are not ready to move from conventional database systems for such reasons as protecting their investment, the immaturity and incompatibility of the new databases systems. Other factors for ruling out such unified systems are the run-time overhead incurred in the additional data-management and/or dynamic binding characteristic of object-oriented languages that may degrade performance to an unacceptable level (when run on current machines). Furthermore, we believe that before such a system matures, more stratified research is required at all the functional layers that compose such a system. Therefore, the idea of one unifying database-programming language remains unrealized in most existing CAD databases. Thus, it seems that "legacy" systems based on procedural languages coupled to conventional database systems will be used for a long time yet [7].

The advantages of user-friendly programming environments based on the object-oriented and persistent paradigms, especially for prototyping [8,9] cannot be dismissed as a temporary means to eliminate the "impedance mismatch" despite the added complexity to the DBMS interface system. This implies adding an interface to the DBMS that creates the desired programming environment and does the "matching up" transparently so that the tool designer has the illusion of a matched environment.

The following section focuses on the functional stratification common in conventional database systems.

1.3 The Conventional Solution - System Stratification

The functionality of the interface system depends on the gap that must be bridged between the DBMS and the tool. In an ideal interface for conventional databases, the functional layers depicted in figure 1.2 may be recognized: The base layer is the operating system. On top rests the DBMS.

Definitions

An application programming interface is a system that enables an application program to transfer data to or from external systems.

From an application programmer’s view-point it is a well defined set of functions that may be called in the program, and a set of data structures for importing or exporting data.
A large-grain object is an (abstract) collection of smaller objects called the fine-grain objects, such that the large-grain object is managed by a (part of a) system as a coherent whole.

The DBMS interfaces with the tool via an application programming interface. Conventional DBMSs vary in the application programming interface functionality. The following combinations are common: only large grain management, only fine grain management and both large grain and fine grain management, the latter being the ideal case. Systems also vary in the interaction that exists between the components. The arrows (in figure 1.2) show the necessary interaction between the different components. A pointer represents the capability of a layer to call functions from the layer pointed at. Stippled pointers indicate a useful but not primary interaction. Systems without such interactions cause restrictions on the functionality. The interface layer may be decomposed into three main parts as depicted in figure 1.3.A: The interface to the tool, the interface to the DBMS and the main body of the interface that is responsible for the two way mapping. The dashed lines imply that the interfaces are embedded in the system. Figure 1.3.B shows a more flexible architecture, in which the application specific part and the DBMS specific part are separate modules. This structure is desirable since it enhances the usability of the interface.
1.4 Requirements

This section provides a list of requirements with regard to the interface system, the DBMS, or the programming environment. The requirement list is usually the first step towards developing a system. It is not feasible to support all the requirements, thus systems differ mainly due to the priority and weight assigned to various requirements.

1. Long term storage - the interface should offer facilities for permanently storing an object in the database.

Our approach towards an interface system is that depicted in plate B of figure 1.3, therefore from the interface system’s viewpoint the database system is ancillary.

The DBMS should be able to manage large grained data as an object. Given the PID, a database object can be fetched or stored on request using the DBMS’s application programming interface. A sensitive issue related to the choice of DBMS is the granularity level it supports. A DBMS performing fine grained data management does not lend itself to clustering (forming clusters), since the data transport is on a per-object basis. The clustering plays a central role in our approach.

Definition
Partitioning is the act of associating the persistent objects resident in working memory to appropriate clusters.

2. The interface should perform partitioning.

If no inter-cluster references exist between objects then the clusters are disjunct and the problem of saving and loading of the objects can be handled easily.
3. Transparency - the interface should trigger loading and saving.
   The programmer needn’t to be aware of an object’s state, e.g. whether it is already volatile, or needs to be loaded, etc.

4. Selective retrieval - an ideal interface should support query facilities for retrieving objects from a database.
   If the DBMS supports large grain management this requirement is partially fulfilled since a query facility should already exist that may access a metadata pool containing information about the database objects and their relationships [10].

5. Software re-use - the interface system implementation should be generic.
   The system should be able to operate with a range of DBMSs, varying from simple storage repositories to complex systems that perform high level management such as versioning, locking etc. The process of incorporating the two systems should be swift and simple.

6. Incremental configuration - the tightness of the tool’s coupling to the DBMS (via the interface) should be controllable.
   During the primary development stages of a tool the efficiency is secondary but special emphasis is laid on communication. When this has been accomplished, an incremental upgrade is desirable to fine-tune the coupling, e.g. enhancing access via an efficient data format or conversion to a standard, defining a more refined partitioning of the data etc.

7. Neutral storage format - different tools interfacing through different interfaces might need access to the same data.
   Database objects may be accessed from different tools using different interfaces. In this case a neutral data format is required for the contents, and translator functions are necessary [11]. This has implications on how the transformation of the objects is performed. In the case of structured data, the conversion is fairly straightforward and elements of a filed data structure can be mapped to the instance variables of a volatile object. When unstructured design data is volatilized, a parser (or lexical analyzer) is required to perform a conversion.

8. Concurrent Data Management
   When tools are running concurrently, they might require access to data from other tools. The tools might be running on the same host or on different machines. Therefore, mechanisms for transporting and sharing data are recommended.

9. Multi-media support - various sorts of data should be manageable.
   The trend towards integration of the design and its (possibly graphical) documentation [12, 13] puts strong requirements on the CAD system or computer aided engineering (CAE)
system, since the design data in such a system is an abstraction over different views of
data, structured or not, hierarchical or flat, in quantities varying from several bytes to
several mega bytes which must be efficiently stored and retrieved.

10. Manual intervention - the system should enable manual coding that overrides automatic
operations when enhanced efficiency is desirable.

Towards the end of the design life cycle of a tool, to reduce the overhead, due to initial
interfacing, it may be necessary to improve run-time performance.

11. Uniform handling of data - the system should enable handling of transient and persistent
data in a uniform manner.

This implies that transient and persistent objects may interact freely with each other and
still be separated when a save occurs. In working memory, objects from different
clusters may interact. However, it should be possible to save clusters separately since in a
multi-user and/or multi-tasking environment, other tools needing the data should be
allowed access as soon as possible. Furthermore, a repeated loading of packages in long
running applications might swamp the available memory or cause thrashing if packages
that are no longer needed aren't released (in analogy with hardware caching systems).

12. Multi-viewed objects - an object may have different facets in different tools, which
implies that the data structure of an object can differ depending on the tool that loaded it.
This problem is addressed in [14].

13. Language independent - the interface should be easily constructed in several commonly
available languages. The system should be based on simple principles that work by using
generic mechanisms common to the traditional languages. Therefore it should not depend
on eccentric features associated with a particular programming language.

14. Testing facility - the programming environment should be supported with a test system to
enable controlled and efficient testing of the tools (or system) under development.

The following section is an example that gives an insight into the functionality provided by
several existing interfaces.

1.5 Example: Some Characteristics of Several Tool Interfaces

There are many attributes that may be associated with a tool interface, whose presence or
absence affect the interface as a whole. Some attributes are contradictory, which implies that
design choices have to be made based on environmental requirements. Consider the example
of four models (named Model 1, 2, 3 and 4) that are related to existing interfaces (familiar to
the author). The characteristics chosen in this example are related mainly to persistency aspects,
as felt by the tool programmer (A, B, C, G, H, J), and the relation these have to performance
attributes (D, E, F, I). This exercise is not intended as an evaluation; Firstly the values are based on a theoretical analysis, not on extensive measurements. Secondly, a priority dimension is missing.

Definition

A framework is a software infrastructure which provides a common operating environment for CAD tools. The framework gives global data management services and maintains information about the design.

Model 1 represents characteristics of the NELSiS framework [15] system. Its advantages are speed and flexibility. Its application programming interface [16] has the capacity to form a substrate for fine-grained interfaces. An overview of this system is given in chapter 3.

Model 2 represents characteristics of the Cad Framework Initiative (CFI) design representation programming interface (DRPI) [17] which is a special purpose interface for a specific engineering application. It has a predefined data schema and specific design data management. The system is referred to in chapters 5, 6, 9 and 10.

Model 3 represents characteristics of the OMS system [18] which is a general purpose interface with a configurable data schema and general purpose design data management.

Model 4 represents characteristics of the OPI system [1] which is targeted for the whole development cycle, from rapid tool prototyping, to finalized tool integration. It has an implicit data schema and high homogeneity of code.

Figure 1.4 compares certain characteristics of the models. These are:

A - Flexibility; increases when the number or complexity of rules that a programmer has to comply with, decreases.
B - Code invisibility; increases when the percentage of integration code decreases.
C - Code simplicity; increases when the amount of functions and data that must be incorporated in the tool decreases.
D - Portability; the ease in which a tool originally developed to interface with a particular model can be modified to interface with other models.
E - Run-time speed; with exception of the I/O speed.
F - I/O speed; speed of the I/O operations on the design data.
G - Data accessibility; possibility of direct access to the design data (C pointers).
H - Homogeneity; similarity of operations and data structures within the application code.
I - Consistency; the extent to which the model can ensure the correctness of the persistent data.
J - Semantic expressiveness; the power of the syntax to define the data schema.

Each but represents a specific characteristic of a model and has a value between one and four while the latter is the most desirable.
Tool interfaces have two facets: the run-time facet and the tool programmer’s facet. In figure 1.5 the run time facet of the four models is outlined. The symbols used represent databases (D), servers (S) and processes (P). Models 1 & 4 keep the design data in the application space; the tool can access the data relatively fast. Models 2 & 3 use a server which implies slower access. In models 1 & 2 the tool code must manage the design data while in models 3 & 4 the management is done by the code supplied with the models. This is represented with the stipple lines.

Figure 1.4. Characteristics of four interface models.

Figure 1.5. Run-time facet of four interface models.
Chapter 2

Virtual Object Interfacing

2.1 Chapter Overview

Figure 1.4 of chapter 1 shows that model 4 puts strong emphasis on the programming aspects of the interfacing by hiding the DBMS related aspects from the application programmer. To accomplish this, various interfacing operations must be managed by the interface. One can ask the following questions:

At what operating level should the automation be implemented (i.e. at the hardware level, operating system level, etc)?

The answer is that the automation may be implemented at various levels, and the choice depends on the subjective environment of the implementor. In the author’s case, it is the level of the programming language.

What are the key operations that must be automated?

The answer is based on studying the persistent paradigm which upholds the concept that persistent data should be managed in a formal and transparent way. The two operations to be automated are saving, and loading. These operations are too complex to be automated easily, and it is necessary to simplify the automation process. The load and save operations may be decomposed into a partition, transport and transform part.

Given the answer to the above two, what methodology should be used to implement the automation?

The answer is that an object-oriented language is a suitable implementation vehicle, since mechanisms to support persistency can easily be added to it. The language also dictates a programming discipline that is necessary to uphold the functional consistency.

This chapter is structured as follows: Section 2.2 discusses persistency problems. Section 2.3 presents an overview of the object-oriented Paradigm, since it is the implementation vehicle of the proposed interface system. Section 2.4 provides a personal chronological perspective on the persistency problem. Section 2.5 provides an overview of several original interface systems that support virtuality. (This chapter has been compiled from [1, 19, 20, 21]).
2.2 Basic Persistency Problems and Solutions

1. **Question:** At what operating level should persistency be implemented?

   Different approaches to implementing persistency are being pursued that vary in their degree of transparency, which to a certain extent is dependent on the system level targeted to provide the persistence support. Figure 2.1 illustrates the levels of implementation. Implementations range from *soft*, beginning with programs written in the application language [22, 23], extensions to existing languages [5, 24], persistent languages [25], database systems supporting persistence [3, 26] and ending with *hard* implementations such as persistent operating systems [27, 28] and persistent machines implemented in hardware [29].

![Diagram showing layers of persistency](image)

**Figure 2.1. Layers of persistency.**

The (VLSI) CAD community has converged towards systems built on UNIX [30] operating system and programmed in C or a C-based language.

For practical reasons, this thesis has been restricted to the study of persistency at the softest layer, namely using a commercially available programming language as the implementation medium. The author's choice of medium was based on two main factors:

- The research environment available to the author (and most interface designers and tool developers) was strongly supportive in the supply of UNIX based work-stations communicating in a network, and in the supply of standard programming languages, thus setting the bottom limit above the operating system level. That left the author with the choice of either a persistent language or extending a standard language, or utilizing a standard language.
- VLSI CAD requires an immediate solution to the problems of tool interfacing, therefore the time needed for a persistent language or an extension to evolve and become commercially standardized was considered too long to be practical.

Apart from providing short-term solutions, performing (applied) research at the two softest levels is useful in providing feedback to programming language developers.

Having provided our reasons for the choice of level, we shall now present the persistency problem at this level.

2. **Question:** What are the key operations that must be automated?

The operation involved in making volatile objects persist, called *save* may be decomposed as follows: The volatile objects must first be *partitioned* into clusters, then *transformed* to a persistent packaged format (imploded), and finally *transported* to persistent storage (exported). The terms *implode* and *export* are used since making data persist in a database is a special case of transporting data out of a process. The opposite operation is *load*; The persistent package is transported into the process (imported), and then transformed to volatile objects (exploded). These (five) operations have clear and distinctive tasks and thus break-down the complexity of the automation problem. This concept is illustrated in figure 2.2. Large and small circles denote clusters and objects respectively. Rectangles denote packages.

![Diagram](image)

**Figure 2.2. Decomposition of saving (plate A) and loading (plate B).**

3. **Question:** When and how is the *save* operation triggered?

Traditionally, the tool programmer writes a code segment that explicitly causes a save operation to be performed on an object. In a persistent system, a mechanism must exist that can somehow deduce when an object should be saved, which is equivalent to
knowing when an object is not needed, or may be needed elsewhere. This is obviously a non trivial problem. Determining whether the object is no longer needed in the tool requires a flow management facility. Determining whether other tools need the object requires an inter-tool communication facility. The trigger is necessary to perform a deduction operation, and based on the result, temporarily disengage the running application from other activities to enable the save.

4. **Question:** When and how is the load operation triggered?

This problem is similar to the previous, however somewhat easier. In general a load should be triggered when an object is needed in the application, which is a technical issue. However, if several processes are running and manipulating shared data, the load is tied to consistency conditions.

5. **Question:** How are the implode and explode transform operations performed?

The volatile structure of an object in the application is generally not suitable for a persistent storage medium, therefore a data structure transformation must be performed. This leads to the questions, what persistent format is to be used and how is the transformation operation incorporated in the system? A typical transformation problem is inter-object references since the volatile pointers must be substituted with symbolic links.

6. **Question:** How and when are the export and import transport operations performed?

Here the problem is how does the persistent system know where the persistent object should be stored, and how does it know the persistent location of an object that must be loaded? Current CAD systems often distribute the data, both logically and physically [31]. Furthermore, having located the object, the system must know whether access to an object is permitted, i.e. may the object be checked-in and checked-out?

7. **Question:** Which language should be chosen for implementation?

The most suitable language for upholding virtuality in objects, is the family of (conventional) object-oriented languages.

Many software engineering related publications, such as [32, 33, 34], promote object-oriented languages, and claim that the software productivity can improve due to better modeling abstraction [35] with only a slight degradation in execution speed.

The next section provides an overview of the object-oriented paradigm, and explains its advantages from the view-point of virtual objects.
2.3 The Object Oriented Paradigm

Terminology

An object-oriented program is a system of interacting objects. The objects encapsulate data known as the object’s members (or instance variables), and the algorithms that specify the manipulation of that data known as the methods. The objects can be manipulated by sending messages. The manipulations directed by the message are carried out by an appropriate method. An abstract data-type, commonly called the class, is a template that defines the properties and behavior of a bag of objects, known as the instances of the class.

Most object-oriented languages have an inheritance mechanism; the member variables and methods of one class, called the base or super class, are included in another class, called the derived or sub class. In turn, a derived class can be the super class of other derived classes, etc. The ability of a derived class to have more than one base class is known as multiple inheritance as opposed to single inheritance when only one base class is permitted.

Dynamic binding means that the implementation of a message is determined at run-time and is dependent on the class of the object that receives the message.

Operations on the objects take place through a well-defined interface. This screens the client program from the actual implementation. Usually, instances are created at run-time. Instances of a class have an identical set of methods, and may differ in the contents of the data they encapsulate. Inheritance encourages software re-use since derived classes inherit all the capabilities of super classes, along the hierarchical chain.

Dynamic binding facilitates software re-use since it enables a loose coupling between the client program and the classes to be manipulated, making the program more flexible and general. Dynamic binding involves a run-time penalty since the function to be performed must be determined at run-time. A method defined in a base class can be neutralized by redefining it in the derived class. The re-use of software is best utilized when the object-oriented language has a library of classes for performing the most general operations. These enable rapid definition of application-specific classes.

Not all object-oriented languages have all the features characterized above; some lack features such as dynamic binding or multiple inheritance. A discussion of what is considered an object-oriented language can be found in [34].

The object-oriented paradigm is attractive because it enables the programmer to think about a program in terms of interacting objects that may perform operations upon request (of another object). This approach differs from classical procedural languages (such as C [36]), which do not emphasize a tight relation between data structures and functions that manipulate these structures [37]. The two most important attributes from our point of view are the notions of
inheritance and messaging. Inheritance is crucial since persistent behavior may be defined in a base class, thus, all (user defined) child classes inherit the parent's persistent characteristics. Messaging is crucial since it implies that an object's data may not be accessed directly (via a pointer), but through a function call.

An object-oriented programming language is suitable for implementing CAD applications as well as desirable persistent operations and therefore provides an ideal medium for implementing virtual objects. A comparable result can be reached by emulating the characteristics of an object-oriented language through a library defined in a standard procedural language (such as Widgets in the X Toolkit [38]) structured in an object-oriented fashion.

Example

![Diagram of interacting objects]

**Figure 2.3. A pool of interacting objects.**

Figure 2.3 gives an example of Objective-C objects. To the left are instances and to the right the corresponding class objects. The instances are referred to by other instances, denoted as incoming arrows to the object's entry point (a small circle). Each instance contains local data (in the example M, 30 & L, 25 are of type character and integer respectively). The top instance refers to its class, which enables the program to determine at run-time what method should be carried out (polymorphism). The top instance also refers to the bottom instance. The class objects contain the data structure templates (rectangles to the left) and the method definitions (to the right). The IS_A variable in an instance refers to the class. The IS_A variable in the (top) class refers to the (bottom) super class. In some languages the class itself is also an operational object, while in others it is just a template.
2.4 A Chronological Perspective

The author's interest in the subject began through studying work carried out by J. Annewinkel [22], concerning development of an Objective-C [39] interface to the IRIS DBMS. This led to the design and implementation of a simple persistent prototype system [20] that validated the power of an object-oriented language to support transparent persistency. Based on this experience a second prototype [21] was conceived that was oriented towards the VLSI CAD environment. The system proposed in this thesis is derived from these prototypes.

2.4.1 The Objective-C Filer
Objective-C is a superset of C. It provides a middle way between Smalltalk's [37] flexibility and C's efficiency, since in time critical situations, one can revert to pure C.

An important feature of Objective-C is its efficient support of polymorphism; object references are defined through the id type, which is a C pointer, that points to the first byte of the object it references. However, the class of an instance is only known at run-time. This makes Objective-C suitable for CAD but has implications on the persistence mechanism. We present a brief summary of both systems to provide insight into the design issues and trade-offs involved.

The Objective-C compiler (1986) is distributed with a class library; among these classes is the filer class, which may be considered a primitive persistency mechanism. The class implements simple load and save operations. By sending an instance the message "storeOn" with the name of a file as argument, the filer mechanism saves the connected component (all objects recursively reachable via pointers from that instance) in the file. The inter-object references are first substituted with links, then each object is transformed to an ASCII representation which is appended to the named file. The load operation is complementary. The method readFrom enables the connected component to be reconstructed in a process.

The system is primitive since any objects referencing the connected component are not "aware" of the filing; the reference cannot be reconstructed, since upon reconstruction a new working memory address is selected; also, the root of the connected component cannot be loaded transparently, since the readFrom message must be sent, to trigger the process. A problem is that the mechanism cannot regulate the amount of objects to be saved or loaded. Another problem is that all connected objects are affected. Problems may occur if only a subset of the volatile objects is to persist. Also, the programmer might only want a subset of a particular object's attributes to persist. The filer doesn't cope with these problems.
Examples

Objective-C Filer save

[a storeOn: fileA];

Objective-C Filer load

[Node readFrom: fileA];

Figure 2.4. Objective-C file load deficiency.

Figure 2.4 illustrates a problem in the load operation. In plate A, instances a, b, c are saved in file A. From object m's view-point instance a remains volatile. Plate B shows a deficiency in the explode operation of load. If the load is performed in the same process, then duplicates of a, b and c are created. To uphold consistency, this implies that objects a, b, and c should be deleted, and m's pointer to a must be reset to a.1 or alternatively, objects a.1, a.2 and a.3 should be reconstructed at the original locations in the process space.

Figure 2.5 illustrates a problem of the file's implode operation. If objects a and d are both saved with a file storeOn operation, a duplicate of object b is created as part of the connected component of a and b. Thus, a duplication of b is created in persistent store. In short, the file can only uphold consistency if the save operation is performed from one root object such that all objects in the process belong to its connected component. Consequently, the load operation must occur once during the initialization phase of a proceeding process which volatilizes the whole connected component.
2.4.2 The Objective-C Iris Interface

The Objective-C Iris interface (1987) enables persistency of simple database objects. Persistent objects are created as instances of the class *PersObj*, but are otherwise indistinguishable from other Objective-C instances. The interface can interact with an existing database, provided the data-schema defines suitable types and functions to be used by the interface. The mapping between IRIS object types and functions is defined in an ancillary file, which may be created and modified interactively. At the lowest level, the interface encapsulates the application programming interface functions to IRIS, called OM, in the form of several Objective-C classes. A database is also represented as an instance of a class, which contains methods for managing persistent objects. Persistent objects are defined dynamically upon creation, and then remain related to a particular database, which will store them if a commit command is given or at the end of the database transaction. An object's behavior is switched from the normal class behavior to the persistent behavior by re-implementation of part of the Objective-C message mechanism, and transparent switching of the IS_A variable of the instance. A major problem is high overhead caused by the fact that the granularity of the objects transported to and from the DBMS is fine. The developer (J. Annevelink) suggested that a clustering mechanism built into the object-oriented interface could reduce this overhead.
Example
Figure 2.6 is an example of a simple program using the programming interface.

```c
main()
{
    id iris_DB, anObject, anotherObject, demoScan;
    int xCoord, yCoord;
    // begin a transaction with the IRIS database called: 'mydb'
    iris_DB = [irisDB begin: "mydb"];
    // create a persistent object that is an instance of the class
    // DemoPoint and store it in database iris_DB
    anObject = [PersObj create: DemoPoint named: iris_DB];
    // set the coordinates of the point
    [anObject x:xCoord y: yCoord];
    // print the point
    [anObject print];
    // retrieve the IRIS DemoPoint whose coordinates
    // are xCoord, yCoord from iris_DB
    demoScan = [iris_DB query: "find p/DemoPoint where
        xLocDemoPoint(p)=%d and yLocDemoPoint(p)=%d
        and yLocDemoPoint(p)=%d,xCoord,yCoord"]; // really get it from the scan object returned by the query
    anotherObject = [[[demoScan next]at: 1] Object];
    // what did we get?
    [anotherObject print];
    // that's enough
    [iris_DB End];
}
```

Figure 2.6. A simple Objective-C Iris interface program.

The author attempted to enhance the concept by introducing partitioning of the data into clusters, thus reducing the DBMS transport frequency. This resulted in two experimental prototypes, ODI#1 and ODI#2 [20, 21]. Both are implemented in Objective-C, but are based on different concepts for implementing persistence.

2.4.3 The ODI#1 Prototype
The first prototype (1988) is based on principles similar to PS-ALGOL (see section 2.5.2); i.e. there is no difference in type between persistent or transient objects and persistent objects are separated from transients by traversing the connected component of a root object. To reduce the amount of PID administration, objects are clustered. The cluster is the unit of data transfer to the back end of the system (a data repository or database system) and supports efficient save to or load from disk (I/O) by avoiding data fragmentation.
Several types of cluster are supported depending on the inter-relation of objects in different clusters. The idea of supporting different cluster types is the efficiency at run time. The more complex the inter-cluster object references, the more complex (and CPU intensive) is the loading or saving of the cluster.

The application designer can enable run-time creation of root objects of any cluster type. In default mode, all clusters use a default type (*Global*), which enables total transparency. This is desirable in first draft prototyping. The programmer of a working application can focus on efficiency by enforcing other cluster types. Partitioning of the connected component of the root object is also done automatically, unless overridden, with cluster size being the major criterion.

This model provides a transparent persistent environment but requires fairly complex mechanisms to maintain consistency. Another problem is that data can only be freed from working memory by first storing the memory resident objects, i.e. saving the whole connected component, like the Smalltalk-80 *snapshot* (section 2.5.1). The advantage is that at run-time little data-management is required, so the application can run efficiently.

**Example**

Figure 2.7 illustrates the interaction between the cluster types. The simplest cluster is called *Private*, since there may be only one entry point (one object that is referenced from other clusters), namely the *root*, and no exit points (all non-root objects in the cluster can reference only internal objects).

![Diagram](image)

**Figure 2.7. Example of clustering in ODI#1.**

The second type is *Entity*, which may have one entry point (similar to *Private*) and also numerous exit points, i.e. objects with inter-cluster references. The third cluster type is *Global* which sets no limits on inter-cluster references, be they incoming or outgoing.
2.4.4 The ODI#2 Prototype

The second prototype (1989) is more oriented towards the CAD environment. The complexity in CAD systems can be handled efficiently by taking advantage of the hierarchical structure of the design data [11].

![Diagram of clustering of CAD objects in ODI#2.]

Each class may be provided with persistent knowledge which is dynamically available to its instances. In this approach the persistence is inherent to the objects. A set of classes provide persistent capability. An application object that is defined as a specialization of a persistent type inherits the persistent capability. Here, a clear distinction is made at compile time between transient and persistent data.

This model also supports clustering but in a different manner. Persistent classes stem from the Private or Shared classes. Instances of classes that inherit Private are always fixed to a cluster, and throughout their lifetime cannot move. They represent internal parts of some abstract object which is the root of the cluster. Instances of classes that inherit Shared can be shared between clusters, i.e. can be referenced from other clusters. The Shared class has one instance variable of type id named persData which is intended for storing persistent information about itself, such as its type and its PID. The type of an object whose class inherits Shared can be changed dynamically.

Three types are supported, Stationary, Migrant and Cluster. By default instances are Stationary, i.e. they belong to one specific cluster but can be referenced from other clusters. They "know" which cluster they belong to, since this data can be derived from the persData instance. Cluster type instances indicate a 1:1 relation with the cluster, i.e. a cluster of objects may only exist on condition that an instance typed cluster exists. The cluster type is responsible for managing all the objects associated with the cluster. Finally Migrant typed instances have the freedom to move from one cluster to another as long as they are referenced from it. This model is more
object-oriented since the objects have more autonomy. This provides flexibility when configuration changes are required since the system is protected through the encapsulation. It is more efficient since the object’s persistent data is stored in the object removing the need for look-up tables. The disadvantage here is that transparency is degraded, since the application programmer must be aware of the clustering to a certain degree and has to code persistent information into the classes. For modeling structured data OD2 is a comfortable environment, but support for unstructured data is not ideal.

Example
In figure 2.8 the parameterized notation X_Y(Z) is used to define the relations between objects: X can be a C for Cluster, S for Shared or P for Private. Y can be A for Associated or F for Floating. Z denotes the owner of the object. For example, P_A(2) implies Private and Associated to object 2. Objects 5, 11 are elements of complex 6 which together with 7 is an element of complex 2. 2 in turn is an element of complex 1. Objects 10, 12 are elements of complex 9 which together with 8 is an element of 4. Complex 4 isn’t an element of a complex object. Object 3 is Floating, i.e. does not have a predefined owner.

2.5 Original Persistent Systems
Atkinson (1974) [3] is a pioneer in persistent programming research; Atkinson’s approach was that persistence should be an orthogonal property of data, independent of data type and the way in which the data is manipulated. A persistent mechanism in an object-oriented environment has two major features; every persistent object has a PID and a dictionary (look-up table) that associates the PID to its physical location. When a message is sent to an object, the object’s location is looked up in the dictionary. If the location is in working memory, the message is passed on to the object. If the object has a secondary storage location it is first read in, the value in the dictionary is updated and the message is passed on to the now memory resident object. This system offers great flexibility since any object can be stored or retrieved independently of others. Both features are problematic; a huge dictionary must be kept in working memory, PID’s become large numbers in big systems causing space overhead, and table look-up on a per message basis creates run-time overhead. A chronological view of the evolution of persistency may be found in [40].

2.5.1 The Smalltalk-80 Snapshot
Smalltalk-80 uses a similar principle in managing memory-resident objects and enables a limited persistence in the form of snapshots which preserve the working environment (called an image) on disc. The system’s main problem is that the environment is closed and almost every object requires a PID and the number of objects is limited in most implementations and run-time speed is degraded due to the look-up procedure.
2.5.2 PS-ALGOL
Definition
An Object heap is an abstraction over (part of) the volatile objects resident in an executing tool.

PS-ALGOL incorporates a hybrid system, in which persistent and transient objects intermesh in the working memory, and get separated during an update operation. A root object defines the point from which all persistent objects are identified by transitive closure of reachability [3]. PS-ALGOL is derived from the strongly typed programming language S-ALGOL. It treats persistence as a property orthogonal to type. Objects from the active object heap can be stored in a database with its type and pointer intact, and then be accessed again by other programs. PS-ALGOL used the concept of reachability as the means of identifying persistent data. The persistent object ID differs from the volatile ID. The system undertakes the responsibility for preserving the persistent data in long term store and organizing the movement of persistent data from disk to active heap. The objects are clustered by type in the database pages to factor out the type description information.

2.5.3 Alltalk
Alltalk extends Smalltalk-80 by providing persistence to objects without adding a database sublanguage. Objects persist in a database on disk, and can be retrieved by an identifier or via a query that specifies a value or range of values for an instance variable. It uses a 32-bit integer to provide objects with a persistent identity which serves as an index in an object table. In this way there is no distinction between volatile and persistent objects. Objects are discarded when no longer referenced. The Alltalk object manager performs persistent services: it obtains a memory pointer given a persistent object pointer called an oop. The interpreter and its primitives may manipulate the object or its class without further calls to the object manager. The object manager can also create new instances, and change the size of an instance. It uses a buffer manager and a pool to manage memory resident objects. The buffer manager uses the database access manager to manage the transfer of objects between memory and disk. Alltalk remains entirely faithful to Smalltalk semantics. The system marks changed objects (modified after the last commit to the database). When the transaction manager is called to do a commit, it writes new objects and existing objects that have been updated to the database. It uses algorithms that prune the transitive closure of the active image, so that few objects are actually examined.

2.5.4 ODE
ODE [42] offers a single integrated data model for both database and general purpose manipulation. The database is defined, queried and manipulated in the database and programming language C++, which extends the object definition facility of C++ [43]. It provides facilities for creating persistent and versioned objects. Persistence is the property of object instances not types, and persistent objects are accessed and manipulated in much the same way as volatile objects. The class facility supports data encapsulation and multiple
inheritance. Persistent objects are allocated/deallocated using the special operators `pnew` (that allocates the object in persistent store) and `pdelete`. Persistent objects are referenced by defining a pointer to a type and a type qualifier (`persistent`) e.g.:

```
persistent point *mypoint;
mypoint = pnew point(initial values);
```

Pointers to persistent objects always refer to persistent objects and ordinary pointers always refer to volatile objects. All persistent objects of the same type are grouped together into a cluster. The name of the cluster is the name of the corresponding type. The ODE be can queried about the clusters that exist in persistent store. Before creating a persistent object, the corresponding cluster must exist, which is created/destroyed by invoking special macros. Persistent objects may have an unlimited number of versions. The current version of an object is updatable, but old versions may be read-only depending upon the implementation.

### 2.5.5 OCT

OCT - Chang and Katz [44] propose run-time clustering and buffering schemes, which suggest an improvement of up to 200% in DBMS response times (based on information collected from the Berkeley CAD group’s OCT design tools), running on top of the object-oriented data manager OCT. The clustering and buffering algorithms can use structural relationships and inheritance semantics. The system distinguishes between static and dynamic clustering. In static clustering the partitioning is done by the database administrator. Dynamic clustering is done at run-time when the concurrent accesses are permitted. Clustering on object creation or update may degrade the response time for writes but can be offset by a large improvement of reader’s response time. The inputs to the clustering algorithm can be user’s hints such as "access by configuration", the inter-object access frequencies, and the characteristics of inherited attributes. The user’s hints are registered into the system through a procedural interface. The clustering algorithm chooses an initial placement for each newly created instance based on which of the instance’s relationships is most frequently traversed. This information is kept in the corresponding data type and is inherited by newly created instances.

The previously presented systems are not readily comparable. First, evaluating systems is beyond the scope of this work due to the labour intensive nature and difficulty of such an undertaking. Secondly, these systems vary with respect to the implementational level as described in figure 2.1. For example, PS-ALGOL may be properly placed at the persistent language level. ODE may be placed at the level of extension to an existing language. Smalltalk-80’s snap-shot capability is too primitive to make it a persistent language. Alltalk may be regarded as a persistent language, with the advantage that it accomplishes this without deviating from the Smalltalk-80 syntax. OCT object management is implemented in the form of a procedural library, and therefore may be placed at the application language level. Two more systems that have received attention from the ECAD community are Objectivity and Ontos.
2.5.6 Objectivity
Objectivity [45] provides a set of database facilities. It has a tool-kit architecture, consisting of several managers, e.g. object manager and type manager. It has a language interface that enables defining object types in C or C++. Two capabilities for data modeling are "varrays" and associations. Varrays are varying-sized arrays of elements. Multiple varrays may be defined in a single object. Associations are directly traversable inter-object links. These two types enable constructing trees, sets and lists. Associations may be unidirectional or bidirectional, and may be of the one-to-one, one-to-many or many-to-many variety. A set of associated objects may act as a single composite object with both data structures and methods. Associations may cross databases, thus they can be used to maintain dependencies, views, and relationships. The DBMS facilities and data cache are located in the application's address space for direct access and perform demand loading. When interfacing, the source code is not completely transparent since special "handle" types are used to define persistent attributes. It supports complex composite objects (hierarchically) and provides a browser for browsing through the object heap.

2.5.7 Ontos
Ontos is an object-oriented DBMS [46]. Ontos is built of layers, each constructed of a set of object classes. Each layer is accessible with an application programming interface. Once an application has been designed and the class structure identified, the developer creates C++ class definitions, which are passed to a utility that automatically generates the database schema (in the database). The base system includes several containers, or aggregate classes which can be used by the developer as building blocks for applications. The aggregate classes are integrated into the object database. Meta classes are instances that provide information on class definitions, including properties or data structures, and operations or methods. The data of these classes enable development of interpretative environments. This capability enables the execution of any C++ function from within an Object SQL query. The meta classes enable support of several property constraints such as "ordered", "indexed", "required", and "unique". Thus a basis for data integrity checking is present. The system runs in a client-server environment and provides distribution of process and data. The object location mechanics are transparent to the application developer. A logical database is represented by a simple string name created by the user. It is probably one of the most advanced virtual object systems currently on the market.
Chapter 3

OPI - an Experimental VOIS

3.1 Chapter Overview

OPI (Object-Oriented Persistent Interface) is an experimental system that we have built to evaluate the virtual object concept, using an object-oriented language as the implementation vehicle. Virtual objects can be transformed and transported transparently between various components of a distributed network, namely tools and DBMS’s. The tool programmer can supply configuration data to guide the transportation.

![Diagram of OPI system](image)

Figure 3.1. Object flow in OPI.
To construct the interface system we must define an architecture that accommodates the functionality required of the system. In our case, this is based on the requirements specified in section 1.4, the selected characteristics of model 4 as defined in section 1.5 and the automation decomposition described in section 2.2. OPI may be viewed as a cache mechanism with object granularity regulation. It partitions memory resident objects in the running application program into clusters. Figure 3.1 provides an overview of the object flow and the functions that the system performs, namely, bidirectional transformation and transportation, and creating, destroying and partitioning clusters.

Several memory resident clustered objects are displayed at the top. The stipple lined rectangles denote clusters. Note that direct pointers exist between objects, since from the programmer's point of view, all objects are virtual. The second level represents OPI's fine grain management activities, namely loading and saving objects. The following questions arise:

What architecture is required to support this functionality?

OPI is composed of several functional modules (managers). The object manager performs the essential run-time operations such as clustering and addressing the "heap" of objects resident in memory. The mobility module is responsible for transformation, transportation and inter-process communication. The partitioning manager is responsible for ensuring that the application objects are associated with the appropriate clusters, based on configuration information. A configuration module is responsible for the configuration data related to the partitioning, the concurrency and fine grained data management.

How is the interface system coupled to the framework?

The solution is to treat OPI as a normal tool coupled to the framework, and apply the standard NELSIS procedure for tool integration [16]. NELSIS is used only as a test vehicle to provide a base for OPI to operate. The procedure may be applied for other DBMS's.

Given the architecture what data structure should be used to support the persistent and mobile characteristics of an object?

The solution lies in identifying an intermediate management layer called the logical layer. This layer manipulates objects that are in a transient situation, i.e. they are either not yet exported from the tool (e.g. to the DBMS) or are not yet exploded in the tool. This stratification is the key to realizing the virtual characteristics.

This chapter is structured as follows: Section 3.2 provides a general view of the architecture of the OPI system decomposed into functional modules. Then a specific view of the coupling with the NELSIS DBMS is provided.

Section 3.3 discusses the virtual object concept, introduces a logical and physical management concept, and illustrated how this enhances virtuality.

Section 3.4 deals with ancillary support: Apart from the VOIS, the OPI programming
environment is enhanced with two systems, the *Configuration Management System* (CMS), that enables configuration of the persistent data, and the *Automatic NELSiS Tester* (ANT) for testing the tools during development. In this configuration all the systems use NELSiS as a base, as though it were the natural extension to the operating system. (This chapter has been compiled from [19, 2]).

3.2 OPI Architecture

3.2.1 Functional Module Overview
Figure 3.2 provides a simplified decomposition of the system into functional modules.

Object Manager
The kernel is the object manager that performs key generic operations such as triggering of persistent activities, maintaining the clustering infra-structure, and managing the persistent and volatile object addressing. The cluster mechanism maintains the virtual object heap. A small set of control functions is available for the tool programmer.

Partition Manager
The partitioning module is dedicated to assigning application objects to the appropriate clusters. The partitioner accesses information stored in a configuration data heap to enhance the partitioning. The partitioning may be controlled by the user (tool programmer) through a file containing configuration data, which specifies the desired inter-relations between CAD object types, and various persistent characteristics. The partition data may be loaded in the tool.

Mobility Manager
The system has a default transform transport and communication mechanism, and is linked to a default DBMS. The transformation module may be extended by the tool programmer to comply with specific DBMS requirements, likewise the transportation module. The transport mechanism interfaces with the DBMS. If the transform or transport mechanisms are insufficient, the methods may be changed by the application programmer. Furthermore, the DBMS may be replaced by a user specified system.
OPI may be extended since applications have different requirements with regard to multi-user access to data, manipulation of data, the kind of data repository used, etc. OPI interfaces with the application program via the library class Virtual which supports the physical facet of the object by supplying the virtual characteristics.

Several DBMS's may be coupled and the system may select the appropriate transform/transport method dynamically by querying the configuration data. The prime requirements of a coupled DBMS are: accepting a package of arbitrary length, generating a key associated with it, retrieving (storing) it, deleting it and releasing the association.

### 3.2.2 The NELYSIS Framework

Specific details such as how the repository is manipulated (interface), what form of keys it uses (data structure), whether they are all memory resident, or hashed via files (in the case of thousands of clusters) etc, vary with the implementation.

The NELYSIS DBMS [47] is implemented in C and runs on UNIX. The framework provides sophisticated large grain data management. It uses a variant of the semantic data-model OTOD [48] (explained in section 4.2). NELYSIS supports hierarchy, view-types, versioning, state management, multi-user facilities, data sharing and distributed design data management.
Definition

A **design object** is the unit of data managed by the DBMS.

In this context it is equivalent to a *database object*. The data is accessed via the Data Management Interface (DMI) [16] which defines a uniform communication protocol between tools and the DBMS, thus upholding data independence. The framework [10] provides the global data management services (figure 3.3).

Definition

**Metadata** is primary information on the presence of design objects, their relationships, their version history, and the operations that have been performed on them.

**NELSIS** maintains *metadata*, rather than operating at the level of the detailed design descriptions. This information may be graphically browsed to view hierarchy, equivalence and various other relations between design data with a tool called DSUI. The raw design data is operated upon by the design tools.

The metadata is collected by the framework while the tools communicate with it to obtain access to the actual design descriptions. One of the major organizing principles in the **NELSIS** framework is the *project*.

Definition

In **NELSIS** a *project* provides a local context for the design activities of one or more designers. It contains a collection of design descriptions that can be operated upon from within the project context.

The framework employs a check-out/check-in transaction model to support design transactions as performed by tools on design objects. The metadata management module (MDM) administers the in-progress transactions and maintain a transaction history. Individual versions are organized by MDM along with such attributes as version-number or version-status. The framework layer interfaces to design tools via the DMI.
3.2.3 Coupling between OPI and NELSIS

Figure 3.4. Coupling between OPI and NELSIS.

Figure 3.4 illustrates the coupling. From the framework view point OPI is just another tool, and communicates via the DMI interface. To the left is a blow-up of a tool. There are 4 interfaces,
the OPI-tool interface, the OPI-OPI server interface, the OPI-DMI interface and the OPI-storage module interface. To the right are two tools working on the same design project denoted PS (project server), T1 running on machine A, and T2 running on machine B. The DMPD is the framework daemon for initiating the NELSS project servers. T2 has access to metadata via shared memory while T1 has access via the server. The triangle denoted P is the persistent data store in which each circle represents an object. Each of these objects (blown-up into triangles denoted C1 and C4) may be accessed directly from the tools. This removes potential bottlenecks, since only the metadata management must use the server.

3.3 Virtual Objects in OPI

In the virtual object paradigm, the application programmer views CAD objects as both virtual and object oriented.

Definitions

Object Mobility implies that an object may be imported and exported between tools or a database.

A Virtual object (in the context of OPI) is an object that has the characteristics of transparent mobility and persistency, where transparency implies that the characteristics are supported automatically (to a large extent).

The view that virtuality embraces more than database interfacing is also described in [49].
Figure 3.5. Object facets in OPI.

Figure 3.5 provides an abstract view of the various facets of a virtual object. At this level of abstraction, objects are regarded as large-grained (composite objects).

Definitions

The **Physical facet** is the level of volatile application objects, i.e. this is what the application programmer sees and controls in the source code. The composite is managed by a **physical manager** which is an object invisible to the programmer. Thus the composite is exploded, and the programmer only sees individual objects.

The **Logical facet** is an intermediate level managed by the virtual system, that regulates the object mobility. A composite is in imploded form, and managed by a **logical manager** which is an object invisible to the programmer.

The **Database facet** is the representation of the composite in packaged form, in the database, managed by the DBMS.

Support of the logical facet dictates the data structure of the interface system (The structure is handled in chapter 4). In conventional systems, there is usually no intermediation between the physical and database facets. Plate A shows the trajectory of a composite object between the various states: The logical level is vital for creating a framework for the automation of persistency; Since it is sandwiched in the middle, it is responsible for the import/export and implode/explode operations.
The logical level is also responsible for the inter-tool mobility of composites. The system must support inter-object references at all levels.

**Definitions**

A *referring object* is an object that contains a reference to another object, the *referenced object*.

A *physical reference* is a pointer from one exploded object to another exploded object in the same process.

The value of a *physical reference* is called the *physical ID* of the referenced object.

A *logical reference* is a relation between a *referring object* and a *referenced object*, on condition that at least one of the objects is not in the physical state. The value of *logical reference* that enables location of a logical object is called the *logical ID*.

In a virtual environment *physical* and *logical* references are managed. Inter object references are automatically maintained disregarding the state of the inter-related objects. In figure 3.5, $P \rightarrow L$ and $L \rightarrow P$ imply an implosion and explosion of the composite in a tool’s process space respectively. $L \rightarrow D$ and $D \rightarrow L$ imply an export from and an import to a tool’s process space respectively.
Examples

In figure 3.6 the various reference types are shown. The solid arrows depict the physical references and the dashed lines the logical. By regarding the logical and database instances of an object as a 1:1 equivalence it is not necessary to consider references to and from database objects in the model. Figure 3.7 illustrates the copy aspect of mobility between two tools (possibly on separate host machines) utilizing the logical level. A (composite) object \( a \) may be copied from a tool running on host \( A \) to a tool running on host \( B \).
Section 3. Virtual Objects in OPI

Figure 3.7. Copying of a composite object between tools.

Figure 3.8 illustrates the *sharing* aspect of mobility between two tools (running on the same host machines), utilizing the logical level. A (composite) object a may be copied from the process space of one tool to shared memory, and directly interact with objects in another tool running on the same machine.

Figure 3.8. Sharing a composite object between tools.

We now present an overview of OPI which is targeted to satisfy the requirements presented in section 2.4 and is our experimental vehicle for studying the concept of virtuality.
3.4 The Application Programming Environment

![Diagram showing the application programmer's environment](image)

**Figure 3.9. The application programmer’s environment.**

Besides the NELSIS framework, the user interacts with three other systems as shown in figure 3.9:

1. **Configuration Management System**
   
   The Configuration Management System (CM) is a set of tools for enabling the configuration of the OPI related aspects of the application. CM covers four main aspects over which the user may exercise control: the clustering configuration, the attribute selection and the transformation and transportation of objects. The Configuration Management User Interface (CMUI) is a graphical user interface tailored to the requirements of the CM (The CM is discussed in section 5.6).

2. **The application programmer may utilize a test system for tools under development. The Automatic NELSIS Test system (ANT) is a general purpose test system [50] that enables testing the tool under construction and certain aspects of the configuration defined by the user in the CM, e.g. it may test an isolated transformation of a particular object or cluster. Chapter 8 is dedicated to this system.**
3. The Application Development system (AD) is the application programmer's plain programming environment. The choice of environment is left to the application programmer; in the most simple case, it would be the UNIX system.

The application program is compiled and linked with the DMI and OPI libraries. The configuration and testing is an iterative process. When the application programmer is satisfied, the CM may be invoked to generate the configuration data. These OPI related data are loaded at run-time into the application executable. The configuration data is converted to initialized data structures. The result is an executable tool, that will make use of the OPI mobility manager to interface with the NELsIS framework during run-time.
Chapter 4

Object Heap Management

4.1 Chapter Overview

The Virtual Object Interface System is constructed of various sub-systems, which must function together and form a coherent data structure. Thus, this chapter focuses on the data structure and operations on this structure. The following problems must be handled:

What method should be used to describe the system in a manner, such that one may get an overview of the whole system?

We (incrementally) construct a data schema that can describe the entities and the relationships in a visual manner.

What form of object management is required from the object manager in order to support the concept of virtuality?

To support virtuality, the OPI manager must track relationships between objects in the running application in such a way that relations between objects in different clusters will be maintained automatically while at the same time the seams between clusters remain invisible.

What form of clustering is required?

The clustering should be hierarchical since this is the natural way to model large CAD designs. Various types of hierarchical objects provide a high degree of flexibility. A composite object is a set of (simple) objects. A complex composite object is a set of composite and simple objects.

Given the clustering requirements, how are they implemented?

Our approach is to support a special attribute in virtual objects that gives a direct link to the management system. This is an enhancement on conventional object-oriented languages which (only) support direct links to the classes.
How are virtual objects imploded and exploded?

A key is identifying three transition states of an object, and enabling objects to
dynamically function in different roles.

How can the transport granularity be disconnected from the transform granularity?

A unique feature is that the system makes a separation between a cluster and package. A
hierarchical cluster may be mapped to one or several packages.

How can consistency be upheld if migration is allowed?

A key to the migration problem is that the address of an object is related to its
hierarchical position, so that if a hierarchical migration occurs, only the top object in the
hierarchy is involved.

This chapter is structured as follows: Section 4.2 is a short overview of the information model.
Section 4.3 focuses on the virtual object management. Section 4.4 discusses object
aggregation. Section 4.5 focuses on the data structure aspects of object transformation. Section
4.6 presents the addressing problem and solution. An overview of the programming interface is
appended. This chapter has been compiled from [19, 20, 21].

4.2 Choosing an Information Data Model

An information model is helpful for system engineers to understand the major entities and
relations in a software system. This approach is proving itself in the CFI organization which
must provide clear and comprehensive descriptions of complex data structures to reduce
communication errors. The question was, which information model to use?

The common model used in ECAD is an entity-relationship model called Express-G. Express-G has a
wide commercial base and various software modeling tools already exist. We shall use a simple
entity relation model which uses a variant of the OTO-D [51] graphical representation. In our
case we shall use the variant to model the static & dynamic properties of the data structure.
This variant which we shall call OIS emphasizes the schematic representation and primary
relations between objects. In the author’s view the variant is more concise than Express-G and
has a clearer graphical representation (a view shared also by [52]).

The world is modeled as objects (entities) which may be types or instances of types. A
primitive type represents a category of atomic objects which are assigned a value. Composite
types have other (composite) types as attributes. Some attributes are private to an object, that is,
they do not relate to any other object. There are two constructs to model relations,
specialization which has the semantics of is_a and attribution, which has the semantics of
has_attributes.
Example
The model has a graphical representation which we shall use extensively: A rectangle represents an object type. A line connecting the corner of an upper object (A) to the corner of a lower object (B) implies that A is a specialization of B. A line connecting the edge of an upper object (B) to the edge of a lower object (C) implies that C is an attribute of B.

![Diagram](image)

Figure 4.1. OIS graphic schema, an example.

Figure 4.1 is an example of the graphical representation. Let b, c, and d be instances of B, C, D, then the following holds: b may only exist if c, and d exist; A c instance may be part of many b instances, but a b instance may only have one instance of c. Internal objects are stippled. If two instances are of the same type they must have a different attribute set.

4.3 Virtual Object Management
In chapters 2 and 3 we concluded that objects may be in a physical or logical state, the programmer always has a seamless environment, and fine-grained objects are clustered. In this section we focus on the problem of how to accomplish this. In section 4.3.1 we shall answer the question, what exactly must be managed. Thereafter, we deal with the problem, what data structure is necessary to uphold virtuality in objects. To provide an answer (section 4.3.2), we first construct a simple data structure, followed by enhancements to represent an object-oriented language. Finally, in section 4.3.3 the structure for supporting virtuality is introduced.
4.3.1 Fundamental Types and Relations

Figure 4.2 shows a schema that contains the entities that must be managed by the virtual system. The interpretation of the schema is as follows: Type Virtual supports the virtual behavior of objects. Since the object-oriented paradigm has been chosen Virtual type is related to a Class. The inheritance mechanism is represented by the OOL-inheritance (object-oriented language) entity. As explained in section 3.3 virtual objects interrelate using normal object-oriented referencing, which explains the OOL-references entity.

RGO and RDO are roles and represent the referring and referenced objects respectively. In an object-oriented language, the Class and OOL-inheritance entities are already managed by the compiler, thus, to support a virtual environment, the virtual objects and the interrelations must be supported.

Requirement 2 of section 1.4 is clustering of fine-grain data which calls for additional management of virtual objects. This is captured by the OPI-relations entity. Type enables defining various aggregation types (discussed in section 4.4). The creation of a "seamless" virtual environment is possible, since the application programmer is free to use the normal referencing method, and the OPI system can transparently perform the aggregation management.
Figure 4.3. OIS schema of common alternative inter-object relationships.

In figure 4.3, for comparison, we present other possibilities. The types Object and Class are shaded to denote that they are the common types. Plate A represents a conventional procedural language, i.e. the manipulation of data in the form of structures, and interaction via pointers. Plate B represents an object-oriented language, since inheritance is supported. Plate C represents a system that manages all relations between objects indirectly, thus, the programmer is not able to use physical pointers between objects and may only define data structures. This approach is commonly used for implementing object-oriented DBMSs and enables powerful relational management (at the price of run-time performance and flexibility). Plate D represents a system similar to that of plate C that supports inheritance as well.
4.3.2 Programming Language Data-Schema

The schema depicted in the previous section is general. The detailed schema should provide as close a mapping as possible to the manner in which the system is to be implemented. The virtual system interacts with the fine-grained object instances of a program which justifies data-structure modeling at the fine-grained level. From here onwards, we shall incrementally construct the global schema that represents essential types and relations of OPI. A complete schema is presented in the appendix to the thesis.

4.3.2.1 A Procedural Programming Language

Figure 4.4. OIS schema for a procedural programming language.

Figure 4.4 provides a schema for implicit data-structures and types that enables modeling of the C language. It supports implicit data, i.e. some types are unknown, and are therefore modeled as instances of another type. The stippled lines and attributes represent internal values of an entity (i.e. not relations). If several internal values are depicted for a composite type then these are related to one another, i.e. they act as one (composite) value. In the figure this prevents type attribute to have a name with different offsets. The figure is structured in two parts; The metadata part describes the inherent aspects of the data, while the data part enables instantiation. Typedef represents any type that may be defined in the language. The attribute entity represents an attribute of a particular composite type, i.e. a data-structure. The attribute
is a part of the data-structure and is of a particular type. The `instance` element represents an instantiation of a particular typedef. The leaf `ram-id` is the instance address in RAM (the working memory address) and the leaf `value` is the value stored at that address. The `attribute-instance` represents an instantiation of an attribute of a particular type of a particular composite `instance`.

Example
Figure 4.5 provides an example. To the left a data-structure is defined, and to the right an instantiation of that structure. Data-structures and their components may be located at an address. `CfidrLib` is a typedef with domain `struct`. The struct contains two attributes of typedef `struct *` named `pointer-to-cfidrList` and two attributes of type `int` named `Integer`. The attributes also have a name (`props`, `cells` etc). One instance of `cfidrLib` is allocated at 1000. Its attribute-instances are allocated at 1000, 1004 etc. Each instance-attribute has a value. The integer attribute-instances contain an integer value, the pointer attribute-instances contain the RAM ID of another instance.

![C data structure definitions and Computer Memory Map](Plate A and Plate B)

**Figure 4.5.** Example for a real-world representation of the schema.

### 4.3.2.2 An OO Programming Language
In an object-oriented language such as C++, the type `class` is a specialized type of data-structure. We want to model the inter-relationships between classes more explicitly, and therefore distinguish the class from type, as depicted in figure 4.6 as the entity `class`. The modeling of implicit class inheritance is represented by the aggregate type `Inheritance` which has a general class that may have several specializations. Each class definition contains its complete set of attributes beginning from the base class, because in working memory a
structure containing the attributes of a class along the complete inheritance chain is contiguously allocated. A method is identifiable by a name which is unique to the class and a sort that reflects one of three types, i.e. private, public or protected as defined in C++ programming language.

**Figure 4.6. OIS schema for an Object Oriented Language.**

### 4.3.3 Supporting Virtuality
To describe virtuality we define an explicit schema because must deal with known classes. The schema must support the requirement (illustrated in figure 3.2) that virtual and transient objects may interrelate. The structure of figure 4.7 shows how this capability is provided. Class system inherits object and virtual inherits system. System contains internal data necessary for maintaining the virtual management.

### 4.4 Object Aggregation

The justification for the differentiation between system and virtual is that the system object is often an instance of an ancillary opt class, which must also possess system object behavior. (The definition of virtual as a specialization of system is justifiable, even though it has no additional attributes, because it has additional methods).
A composite aggregation represents an attachment of one or more objects to another object. A question regarding object aggregation is: How to associate the object to the aggregate? A bidirectional link enables better management, at the price of both memory and disk space. This is dealt with in section 4.4.1. Having established the aggregation method, another problem is: What kind of aggregation is required? We handle this in section 4.4.2.

4.4.1 Supporting Aggregation

The aggregation link is accomplished by defining a pointer (called mySystem) in the system class. This enables aggregation to be a characteristic of every object (figure 4.8).

**Definitions**

A simple object is a virtual object that is not an aggregate of other virtual objects.

A composite object is a virtual object that is an aggregate of other virtual objects.

Apart from upholding the attachment link, a virtual object may interact directly with the interface system, for example, VLSI CAD tools often have problems dealing with objects from more than one database. In figure 4.7, the (OIS) role-attribute composite of system models the system pointer. The attribute upholds that any system object belongs to only one composite, and a composite contains at least one object.
Example
Figure 4.9 is an example of three object instances in a composite. System pointers are stippled, the rest are user-defined pointers. Object a is the composite root.

![Composite Object - exploded](image)

**Figure 4.9.** Composite-object aggregation - an example.

4.4.2 The Clustering Scheme
Various clustering schemes are possible. There are two aspects; depth and inter-cluster relation. In the simplest case, there is no depth, i.e. a cluster cannot contain another cluster, and no interrelations between objects in separate clusters. In the complex case, depth can be unlimited and any object may interrelate. In some cases, interrelation restrictions may exist, (e.g. conformance to a tree structure). In our approach, OPI allows the programmer to have the most flexible structure, since it best supports the ECAD situation.
Definition
A Complex composite object (cluster) can recursively contain other composite and simple objects, and the composites are not limited to having one entry or exit point. The interrelations are not limited to structural constraints; Any (simple) object can reference and be referenced by any other (simple) object.

4.4.3 Upholding the Clustering Scheme
Having decided to provide the most flexible mechanism, the question is how should it be realized. Our approach is to create a simple tree structure that is visible only to the OPI manager.

Definition
The composite object representing the cluster is a cluster object, however, to distinguish between a cluster abstraction and the actual cluster object, we alternatively use the term root.

A cluster represents the import/export object unit, and therefore contains additional management facilities. A cluster is managed by the OPI system class cluster-manager as depicted in figure 4.10. The cluster-manager is responsible for the transport and transform operations on the cluster. A one-to-one relation exists between the cluster-manager and the root object. We must make a note since this is not shown in the schema. If a system instance is a root then its system-pointer always points to its cluster manager. The result is that given a cluster, there is only one system pointer that may cross its boundary, that of the cluster-manager. This ensures that a cluster may be managed as a unit. If the cluster is in the logical state, the cluster-manager itself represents the cluster (see figure 3.5). As stated in chapter 3, the Logical specialization is the logical object for managing an imploded composite, and the transform and transport operations. The Physical specialization manages the exploded
composite. The cluster-instance attribute is not modeled as a relation to system. The reason is that if the cluster is imploded then the cluster-instance will be (a pointer to) the imploded image (e.g. a string).

Examples
Figure 4.11 illustrates a complex composite object.

![Diagram of complex composite object](image)

**Figure 4.11. An example of a complex composite object (cluster).**

### 4.4.4 Cluster Hierarchy
Hierarchical modeling of clusters enables a good mapping for large designs which are generally hierarchical. Any imploded composite object can be stored independently of other composite objects. When a programmer integrates a program with a virtual object interface (VOIS), two interfaces are formed: the tool VOIS and the VOIS DBMS interfaces. The ideal granularity of data for efficient management that passes through these two interfaces is different. The DBMS side should perform efficient I/O, i.e. relatively large chunks of sequential data, which we call packages. The tool side reflects the designer’s view. Shells of nested objects that can be "zoomed" in specific places when more data needs to be revealed, which suggests a finer level of granularity. Therefore not all composite objects in an imported package need to be transformed.

**Example**
Figure 4.12 is an example of three hierarchical levels of clustering. Objects a, e and h are roots, e is a child of a and h is a child of e. Cluster e also has an imported-imploded child cluster. Note that the seamless environment concept is maintained since the physical object's system object is the candidate composite of the root had there been no cluster definition at that level. This example explains why transform and transport operations are controlled in the logical class.
Figure 4.12. An example of a hierarchical composite.

For example, in the NELSI CAD framework, the data of a design-object may be split into different streams which enables a finer grained check-in and check-out of design-data.

4.5 Aspects of Dynamic Object Management

4.5.1 Transformation

For the environment to be seamless, objects must be loaded into the application on demand at run-time, and unloaded when not needed or required else-where. The mechanism must be coherent with the hierarchical clustering structure. The solution to this problem is to move the cluster-manager up to the level of the parent cluster. (In figure 4.12 for simplification the manager was associated with its own cluster). It may function as an abstraction of the cluster since it is always present. (The actual object may still be in persistent form).

Example

Figure 4.13. explains the implosion (explosion) mechanism: The lowest placed cluster (a) which is initially exploded and managed by a physical object, is transformed to a logical object.
This causes the physical manager to be replaced by a logical manager, which points to the imploded cluster. Next, the cluster with root d managed by physical D is imploded.

Figure 4.13. The transformation mechanism - an example.

The manager A is imploded along with the other objects in the cluster, and D is replaced by a logical, pointing to the imploded string that now contains A. There are two possibilities regarding the imploded data of A. It could be saved in the data-base, in which case A would contain the PID of the package. In the other case it is appended to object A, and flattened within logical object D. Naturally, the explosion works the other way round. For example, let us assume that D is exploded, and A is logical. Exploding A implies transforming the string into allocated objects (as in the readFrom operation of the filer, section 2.4.1).

4.5.2 Physical Referencing

Definition
A static pointer is the address of the physical location of an object in memory.

Problem Definition
Our implementation is targeted for languages with static pointers. For example in Smalltalk-80 a global object table is used (dynamic), while Objective-C uses C pointers (static). If the voix implementor chooses for static pointers, the problems of virtuality are more complex, since the object is physically pointed at. The advantage of indirect pointing is that an object may be
Section 5. Aspects of Dynamic Object Management

referenced without physically existing in the process. Currently, in VLSI CAD, commercial
products with a run-time performance slightly "above C" are at a disadvantage. Presently C++
is the most widely used object-oriented language for ECAD.

4.5.3 Inter-Composite States

In OPI the unit of transformation is the composite object, therefore consistency implies being
able to implode or explode a composite without disrupting the object heap. The problem
becomes clear when one analyzes the possible interactions between composites. Consider two
composites. Since each composite may be imploded (logical) or exploded (physical) four
possibilities may arise:

State S1 represents two composites in logical form.
State S3 represents composites A and B in physical and logical form respectively.
State S2 represents composites A and B in logical and physical form respectively.
State S4 represents two composites in physical form.

The difference between the states S2, and S3 lies in the direction of the pointer between
composites A and B. To uphold the seamless environment, in figure 4.14 all four transitions
between mutual states (solid line arrows) should be possible.

![Figure 4.14. State transitions of two composites.](image)

The transitions S1→S4 and S4→S1 (stipple line arrows) are considered to be redundant since
on a sequential machine only one composite is imploded or exploded at a given time, therefore,
S1→S4 may be composed of the sequence S1→S3→S4 or S1→S2→S4 and likewise S4→S1
may be composed of the sequence S4→S2→S1 or S4→S3→S1. Analogical reasoning rules out
S2→S3 and S3→S2.

S1→S3 or S3→S1 are not really problematic since there is no need to create or break a seam
between the composites A and B. The same holds for S1→S2 or S2→S1.

Thus S2→S4, S4→S2, S3→S4 and S4→S3 remain.
Example
Figure 4.15 is an example that analyses how the various states apply to the interaction between two simple composite objects.

Figure 4.15. Mutual state combinations of two composite objects.

S2 → S4:
The problem is straightforward. Given that object c contains the logical reference of object e, and it may be converted into the physical address, then c may be connected to e.

S4 → S2:
The problem is mild. Given that object e can obtain its logical reference, this may be returned to c, which may then be imploded.

S3 → S4:
The problem here is more complex. If in state S3 object c has received a message that relates to the pointer to e (which is non existent as a physical pointer), then an inconsistency may arise.
Therefore the triggering of a transition must be performed. The implementation of the trigger influences the extent of cascading, i.e. the importing or exploding of possibly irrelevant data.

**S4→S3:**

This is more problematic than S4→S2, since c is not aware of the fact that e has been imploded. Therefore, if c sends a message to e or must convey e’s physical ID, consistency must still be retained.

**Solution**

The solution lies in recognizing two basic statuses that an object may acquire: roles and states, and providing management of these statuses.

**Definitions**

A **role** is a status assigned to a virtual object that designates a particular type of functionality that the object may perform.

A **state** is a physical form of a virtual object that depends on possible forms of the data contained by the virtual object.

A mechanism in OPI enables a virtual object to dynamically and simultaneously take on several roles, depending on its inter-composite interaction. Another mechanism dynamically determines the internal state of an object at run-time. Role and state are attributes of any system object (figure 4.7). Sections 4.5.4, 4.5.5 and 4.5.6 introduce the role mechanism, the state mechanism and the combination of both mechanisms respectively.

**4.5.4 Object Roles** Let us denote **so** as the role **System Object** (section 4.4.1) which represents the local system as observed by an application (user) object and may supply system related information (such as determining an object’s logical reference).

Let us denote **vo** as the role **Virtual Object**. A virtual object may function as an so, e.g. the root object of a composite object.

Let us denote **ro** as the role **Reference Object**. An ro is a role that may be taken on by the physical object referenced, and encapsulates the data necessary for locating the logical object it represents.

When a physical object implodes, it requests all objects that it references physically, to return a logical reference. The reference object is responsible to locate the logical object, ensure the physical object is allocated and return the physical pointer. A reference object must be able to allocate the physical object and return the physical reference without having to invoke an operation that fetches the logical object. Apart from reducing data import, it guarantees atomicity since exploding an object does not effect the objects it references.

**4.5.5 Internal States of an Object**

We distinguish between two kinds of messages that can be sent to an object. An object undergoes state transitions depending on how it interacts with the environment.
Definitions

A system message is a message sent by the OPI system.

An application message is a message programmed in the application.

An Empty object is a physical object whose instance variables are empty (except for the system pointer slot, which points to a system object).

A Full object is a physical object whose instance variables contain valid data, excluding references. Valid data is the data intended by the programmer.

A Connected object is a physical object whose instance variables contain valid data (including references).

We choose to model the state-transitions using the Petri-net model described in [53]. Due to the various variants, we shall explain the mechanism. In figure 4.16 a part of a petri-net is shown. A circle denotes a state within a system. A token placed in a circle denotes that the system is in that state. A system may be in one or more states at a given moment. An arrow from a circle to a bar represents the direction of transition. If all circles pointing to a bar contain a token a transition will occur and a token will be placed on all circles that are pointed to from the bar. For example, if a token is present in the states named $Mc$ and $E$, a transition will occur, and a token will (again) be placed in $Mc$ and in some other state.

![Figure 4.16. Partial Petri-net presentation.](image)

In our case, this standard presentation does not provide a clear schematic, therefore we use a colored Petri-net variant. In figure 4.17 the states $Mc$ and $Mf$ are combined to one state $M$ and the bar leading from $E$ is split-up to distinguish between $Mc$ and $Mf$. The intention of the figure is to clarify the relation between state transition and message categories that cause a transformation.

In each state different messages trigger different transitions, thus the following categorization may be made. The three circles to the right represent an object’s state; A property of this Petri-net is that the object states are disjunct. $E$ is empty, $F$ is full and $C$ is connected. One of the circles contains a token representing the current state of the object. The circle denoted $M$ represents the message to be processed by the object. When a message is sent to the object, the succeeding state is determined according to the current object’s state and the category of the message. The object states are Empty, Full and Connected:
If the object is in the empty state it may change in state, e.g. category f (forward) or remain in the same state, category r (remain). Any application message sent will trigger the object to first transform to the full state, category c (cascade) and resend the application message to itself, which in turn will cause it to cascade to state connected.

Let us suppose the object is in the full state which occurs for example in the case of an import. The data has been retrieved from the logical object. Any references to other composite objects are represented by a logical reference. Categories f, b, c, r switch the state forward, backward, cause it to cascade or to remain in the full state.

Let us suppose that the object is in the connected state. All logical references have been reset to physical. The object is completely integrated in the application, and can behave normally. Category b causes a switch back to the full state in the case of an implosion.

This categorization is only of importance for the implementation of the VOIS system. The application programmer is not confronted with this problem, since application messages are of category c.

4.5.6 Combining Roles and States
This section is quite detailed, however useful for anyone undertaking to implement such a mechanism. The reader may skip the section without loss of continuity.
For convention, we use a parametrized notation for the role and a state of an object, thus:

\[ N/R/S \]

\( N \) is the object’s name, \( R \) is the role (VO, RO, SO), and \( S \) is the state, (E, F, C). The following example exercises all allowed combinations.

\[ \text{PLATE A} \quad S3 \]

\[ \text{PLATE B} \quad S3 \rightarrow S4 \]

\[ \text{PLATE C} \quad S4 \]

\[ \text{PLATE D} \quad S4 \rightarrow S3 \]

\[ \begin{array}{c}
\text{LEGEND} \\
\triangle \quad \text{System Object} \\
\square \quad \text{Reference Object} \\
\cdot \quad \text{System Pointer} \\
\circ \quad \text{Virtual Object} \\
\rightarrow \rightarrow \rightarrow \text{Physical Reference}
\end{array} \]

\textbf{Figure 4.18. S3→S4→S3 transition of two composites.}

Figure 4.18 provides a detailed view of the (problematic) state transitions S3→S4→S3.

\( VO, SO \) and \( RO \) object roles are represented as circles, triangles and squares respectively. At the top (a) shows a more detailed version of composite A (from figure 4.15). Object \( a \) is in the dual role of VO and SO and is Connected. Object \( b \) is a VO and Connected. Object \( c \) is in the Full state. Object names tagged with ‘(‘) play the role of ROs, and ‘(‘) play the role of SOSs. The same instance can play multiple roles. The reference to object \( e \) from \( c \) is logical, in the form of object \( e' \). The pointer from \( e' \) towards \( c \) is a pointer to the slot containing the value \( e' \) (represented in \( c \) as a little circle). This pointer enables a change of the pointer from \( e' \) to \( e \) (if
necessary). $e'$ can communicate with the system and therefore can obtain the real object it represents. If $c$ is sent a message, it changes to Connected. The actual transition occurs through an intermediate phase, shown in (b). The object $e$ is allocated and set to Empty. This doesn't require locating object $e$, which might still be in secondary storage. The pointer from $c$ to $e'$ is replaced with a pointer to $e$. $e'$ now functions as $e$'s so. In this state, if $e$ isn't sent a message, it doesn't need to be exploded. If $e$ is sent a message, state S4 is reached as shown in (c). $e'$'s composite object is located, its objects are exploded and integrated into the application. Finally, (d) shows how the system returns to state S3. Composite object $B$ is imploded, but $e$ is left (not freed) in the Empty state. The reference object $e'$ (a new one) enables composite $B$ to be located if $e$ were sent a message.

4.6 Virtual Object Addressing

4.6.1 Basic Addressing Mechanism
Generating a logical ID for an object is necessary when at least one referring instance is imploded. This ID enables re-location when the referring is exploded. The referred object might be physical, logical or database.

Definitions
A database cluster is a cluster that coincides with a package.
The Universe object is the top most object in the hierarchy of composites. The universe is an abstraction over all objects reachable from the universe object.

We have assumed a 1:1 relationship between a package and a database object (section 3.2), thus such a cluster may be identified using the unique PID of the package. Since all other clusters are contained (hierarchically) within database clusters, and a logical layer is supported, we may confine the scope of the problem to objects in either physical or logical state which must be locatable from anywhere in the universe.

The physical location of an object is simply the RAM address of the object; the logical location is a combination of the physical location of the logical object and local address within the logical object. An addressing system must generate addresses, and given an address, find the location regardless of object state. The question is; what is this logical ID? The sensible approach is to make use of the hierarchical structure. The referred object is always associated with a composite and a composite always exists within a cluster.
Definition
Within a composite, an object has a unique identification called the local- ID (which is simply an integer).

A composite (contains a dictionary that) associates a local- ID with the object's location. When a composite is imploded, the association between an object's local- ID and its address is preserved, and reconstructed when it is exploded. This is a simple filing procedure (as explained in section 2.5.1). This functionality is already depicted in the schema, in the attribute composite of system (figure 4.8). We have now simplified the problem to the scope of physical objects. At this point we have a solution for locating an object within a composite, given that the composite is known. Let us assume that a cluster-manager contains a similar dictionary containing all the composites (This is not necessarily equivalent to the dictionary of the root composite.) then any composite in a cluster is locatable. This mechanism is illustrated in figure 4.19. In plate A an object a sends a message to an object b which has the reference role, and is in empty state. This triggers a demand load; Via the logical address, the object's cluster is located. Let us assume that the composite containing b is still logical. In plate B, composite y is exploded, object b's data is filled and b is reset to state full. During explosion, the dictionary is reconstructed; thus object b is locatable.

The universe ID of an object is a compound ID, consisting of the sequence of logical ID's of all the composites along the path from the universe object to the object, and finally the object's local logical- ID.

Since any cluster is itself a composite, any cluster within a cluster is locatable. This is recursive, beginning from the universe. As long as this cluster is by default locatable, all
objects (in the universe) are locatable. Figure 4.20 shows the inclusion of the addressing mechanism in the schema. Via the *composites* relation the cluster-manager contains the list of its composites. In the information schema, the logical ID attribute of the composites relation is the local path of the composite with respect to the cluster root. The *id* defined as an attribute in the object itself is always the local ID within a composite. A constraint on the composites relation table is that the *logical ID* instances are unique, and the system's (OIS) role is *composite*.

![Diagram](image_url)

*Figure 4.20. OIS including addressing mechanism.*

### 4.6.2 Object Migration

Suppose an object is to switch its aggregate. This has consequences for any referring objects that are still in logical form, since they still contain the previous address of the object. Obviously, a redirection mechanism is required. In the spirit of hierarchical clustering, and local management, it is straight-forward to localize the redirection. Thus, the simplest solution is to place the object's new logical ID in the value slot of the dictionary. However, if we switch the system pointer of a composite, or a cluster, then what is the consequence for the descendents? If we view the world as hierarchical composites, then consistency of logic implies that switching a composite from one cluster to another will apply to all internal aggregates, recursively. This in turn will impact the logical ID of all descendents. Recursively traversing the descendents and switching each logical ID in-turn is extremely unattractive. Actually, we should like to register one change only, at the point where the switch occurred.

**Property:** It is only necessary to redirect the address at the point of switch.

**Proof:** Suppose that a composite with logical ID U..x has been migrated, and a logical reference U..x..y is to be located. The algorithm traverses the dictionary of each composite from the universe, until it reaches U..x, where the redirection value U..m is found. The redirected object is to be found at U..m..y, since no system pointers have been changed except the object with
logical ID U_.X, thus the relative path remains unchanged. Q.E.D.

It is important to note that this method is consistent as long as a generated local ID is never recycled, and the slot with the redirection is preserved. Another constraint is that we assume that a system-pointer of a composite will not be switched to an object that is an internal component of the composite. Upholding this is complicated, while it is not logically sound (in modeling terms), therefore, it is not allowed.

What is the consequence of not being able to re-use an ID within a composite? Theoretically, this is a problem, since the range is limited. In practice, we ignore this since the generation is distributed among all the composites. If we take an integer of 32 bits to represent the local ID, then it is not likely that one composite should generate all possible values. It is known that the Smalltalk-80 environment suffers from such a problem (which restricts its use), however, this is due to the use of a global object table, for all objects in the universe.

What is the consequence of the system accumulating slots with redirections? This is a more serious problem, since each slot consumes space; However, since the tables are local, only tables of imported clusters are in working memory. It is at this stage not known what the frequency of migration is. If the number of migrations required in order to accomplish a VLSI design is negligible compared to the total amount of design data then it is of little use to rectify this problem except from an academic point of view. It is possible to perform garbage collection on redirections, if the whole universe is imported and exploded, since then a referring object will acquire the newest logical ID of a referred object.

Appendix 4.1 The Programming Interface

Appendix 4.1.1 Transaction protocol.
The transaction protocol shown in figure 4.21 (based on the schema presented in [11]) illustrates the major sequence of operations from a running tool’s point of view.
The vertical levels represent the transaction hierarchy. A horizontal line (of rectangles) represents (from left to right) the normal sequence of a transaction at a particular level. Any number of sub-level sequences may occur per level. In the case of newly created objects, the transaction sequence only implies the right hand part. A transaction may be triggered by a virtual user-object (which explains the line between IP transaction to Object transaction). For example, when a new object is created, a composite transaction will only occur when the object is to be saved, in which case it will first be aggregated to a composite, and then imploded; Thereafter a cluster transaction will occur, finalized by the cluster being exported.

Appendix 4.1.2 Main functions
The functions here are described as implemented in C++.

opITransaction

Syntax

aBoolean = opITransaction(aMode, aToolName);

Description
aMode has several options which may be ANDeD:
• INIT: causes initialization of the OPI infrastructure in a tool process.
• QUIT: causes a disciplined shut down of the OPI system.
• SAVE: causes a save of all the (persistent) objects present in the OPI object heap.
• PURGE: causes a purge of all the (persistent) objects present in the OPI object heap.
Comments
This is a function (not a method, since there is no instance to which a message may be sent). Initializing the infrastructure includes creating factory objects for the classes (both opi and application), and creating special cluster objects such as the Universe. Combining several options (by ANDing) is possible, e.g. all the heap objects may be saved (even if they are distributed over several databases) with the option SAVE&QUIT. A QUIT always invokes a purge of the complete heap. aToolName is optional, and used in conjunction with the INIT mode.

opIInitializeDb

Syntax
aDb = opIInitializeDb(aMode, aVirtual, aDBName, aFunctionPtr, arglist);

Description
This is a function which initializes a database interaction. A database instance is returned, which may be of any class that descends from (including) opiVirtual.

aMode may be a combination of ( ORed and ANDed) the following options:
• CREATE: create database (database must not exist).
• OPEN: open database (database must exist). aVirtual is the user object associated with the database. An association is automatically made between this object and the database (key).

Comments
An operation is necessary to establish a relation between an actual database and the opi management system. Most conventional databases are created and opened using a C procedural interface function call, therefore the input variable aFunction is a (pointer to a) function for creating the database, followed by the necessary arguments (argList). Multiple databases of the same or different types may be open in the application. Some combination examples:
• CREATE&OPEN: implies that a database must first be created.
• CREATE&OPEN: implies that prior to opening a database, if it doesn’t exist it is first created.

opITransaction

Syntax
bool = anObject->opITransaction(aMode,aFunctionPtr,arglist);

Description
This method performs an opi system transaction relating to the receiver (anObject). This message may be sent to any object descending from opiVirtual. It returns TRUE for successful completion.
**aMode** may be (an **anded** combination of) the following options:

- **SAVE**: the cluster that contains the specified object is saved in its associated database. If the object is a database object, then all objects associated with the database are saved.
- **PURGE**: the object is purged. If the object is a database instance, and the option is **PURGE&HIERARCHICAL**, objects associated with the database are also purged.
- **CLOSE**: transactions with the database are stopped (relevant only for database objects).
- **HIERARCHICAL**: the transaction must be applied hierarchically to the object.

If the object is a cluster object (root), **floating** objects that are referred (possibly indirectly) by other objects will be attached to the most suitable cluster. Partitioning will be performed if necessary.

Purged objects are not freed from memory, but are given a purge status. A **PURGE** on a database instance will free the memory used by the purged objects. A purged object that receives a (user) message will generate an error. A **SAVE** creates a persistent image of the object (hierarchically). The objects remain in transformed (logical) form.

**Comments**

The ** opiTransaction ** is always applied to an existing instance, and therefore defined as a method. By using polymorphism, there is no need to define separate messages for a database instance or any other instance, although they react differently. The method can be embedded in a user-defined **database** class object and triggered indirectly, rather than appear in the main function of a tool. ** ANDing ** the various modes provides the user with maximum control. For example, the mode could be **SAVE&CLOSE&PURGE**.

The purge function is necessary, since when an object is deleted it must also be neatly disconnected from the ** opi ** system infrastructure. The mode **PURGE&HIERARCHICAL** performs a complex task, that otherwise requires intricate programming. The purge implies totally removing the object from the heap. It is not removed from persistent store.

The **SAVE** function is required usually at the end of a process execution, however sometimes intermediate saves are required. A save causes an implode and export (and therefore operates on a whole cluster). The mode **SAVE&HIERARCHICAL** causes a hierarchical transport of clusters, such that all child clusters are included. If no purge is called for, all the transformed objects are left in logical form. If the object is sent a message it will be transformed transparently. The operations import and explode occur through demand loading (there is no **LOAD** option). The **INIT** could be triggered transparently, however, ** opi ** supports the entire development cycle of a tool. In the beginning the programmer may only want to manipulate volatile objects without any problems that might be connected to the ** opi ** system infrastructure.
Appendix 4.1.3 Ancillary functions.

** opiSetSystemPointer:**

**Syntax**

```c
anObj->opiSetSystemPointer (object);
```

**Description**
Set the system pointer of an object to another object. This is analogous to setting the ownership of an object, however, the process is changeable.

**Comments**
The `SetSystemPointer` enables the advanced programmer to interact with the OPI system, and it enables "hard-wiring" of objects to clusters, which enables run-time speeds close to a procedural interface.

** opiMakeRootObject:**

**Syntax**

```c
anObj-> opiMakeRootObject();
```

**Description**
Notify OPI system that the object is a root. This causes a cluster to be created and incorporated in the infra-structure.

**Comments**
This enables coding the fact that an object is the root, rather than defining this in the configuration data.

** opiTransport**

**Syntax**

```c
aMessage = anObj-> opiTransport(aMode,aDestination);
```

**Description**
aMode may be one of the following:

- **EXPORT:** implies imploding anObj (hierarchically) to logical format and sending it to another destination. The destination is a class, that encapsulates the destination aspects, i.e. another tool or shared memory location etc.
- **IMPORT:** implies that a (hierarchical) object is received from a destination.
Appendix 4.1. The Programming Interface

Comments

SEND and RECEIVE enable the programmer to transport (hierarchical) objects between one tool and another. The send function automatically implodes the object and sends it to the destination. The receive functions performs the complementary operation.

\textbf{opTagStorage}

\textit{Syntax}

\begin{verbatim}
  aBool = anObj->opTagStorage(aMode, tagName, tagValue);
\end{verbatim}

\textit{Description}

\textit{aMode} can have the following options:

- \textbf{INSERT}: insert a tag, named \textit{tagName} with value \textit{tagValue}.
- \textbf{GET}: get the value of a tag named \textit{tagName}. The value is put in \textit{tagValue}. If not found, \textit{FALSE} is returned.
- \textbf{DELETE}: delete a tag named \textit{tagName}. If not found, \textit{FALSE} is returned.
- \textbf{EXISTS}: check if a tag named \textit{tagName} exists, if so returns \textit{TRUE}.

\textit{Comments}

The four options enable manipulation of a mini volatile storage-repository which any virtual object may locally create. This was initially supported to store ancillary volatile object management data locally. Due to its usefulness, the system is made available for the application programmer as well.

Appendix 4.2 A Variant Transition Mechanism

Figure 4.22 is a variant on the mechanism presented in section 4.5.6. Here, instead of using a foreign reference object as an intermediate, in the case of forward reference, the referenced object itself acts as the reference. The condition for this to occur, is that the reference contains additional information, namely the size of the object, thus the reference may be filled with the object's data. An advantage of this variant, is that there is no need to create a foreign object. A disadvantage is that references take up more space in persistent as well as volatile form (in the OPI implementation the variant is used). For example, object \textit{g} in figure 4.22 has a reference to (an object in) the as yet imploded cluster. To complete the picture, an instance of the referred class is allocated, and actually pointed at by \textit{g}. This instance's role is \textit{Reference}, its state is \textit{Empty} and its system pointer points to the logical instance.
Figure 4.22. S3→S4→S3 variant transition of two composites.
Chapter 5

Partitioning Management

5.1 Chapter Overview

In OpTI, partitioning can be controlled. The programmer may first focus on general application aspects of the tool, and then gradually shift the focus towards the DBMS coupling. When the tool has stabilized, efficiency can be enhanced by hard-wiring ownership relations between objects in the code, which is close (in speed) to the conventional tool-database coupling method.

How can the application programmer control the partitioning?

The application programmer can control the partitioning in two ways: One way is to define partition and relational attributes in the configuration data file that is loaded in the tool at run-time. The other way is to hard-wire relationships using a method provided by the virtual class, that enables the programmer to define an object's owner. This last method is efficient and is best suited for mature tools.

How does OpTI enforce the partitioning at run time?

Three partition algorithms may be invoked depending on the situation. If no configuration data is given at all, then all virtual objects are clustered together. Alternatively, virtual root objects can be hard-wired by the programmer, in which case virtual objects referenced by these root objects are clustered with them. If the number of virtual objects that must be associated with a cluster is large, an algorithm is invoked that traverses the complete heap, and based on the configuration data partitions the heap. If the number of virtual objects that must be associated with a cluster is relatively small, then an algorithm is invoked that attempts to perform a partitioning based on elimination operations, to avoid traversing the whole heap.

This chapter is structured as follows: Section 2 states the problems and discusses the requirements of the solutions. Section 3 discusses the partitioning principles, i.e. the solution to the problem related to the data configuration. Section 4 discusses the solution to the run-time partitioning problem and the resulting partitioning algorithms. Appendix 1 adds the partitioning definitions to the part of object addressing. Appendix 2 discusses implementation details and appendix 3 provides a C++ implementation of a partition algorithm. (This chapter has been compiled from [54, 2]).
5.2 Refinement of the Problems Concerning Data Configuration

5.2.1 Requirements Related to Partitioning
(1) An application programmer need not have any knowledge of the framework tool interface. This enables the programmer to focus on the problem at hand without contemplating the framework interaction.
(2) The responsibility for defining run-time persistent data management is removed from the programmer through the existence of a dedicated component in the tool interface. This reduces the amount of code dedicated to persistent data management.
(3) All objects in a process are homogeneous to the programmer be they persistent or transient. This is the concept of a persistent interface as discussed in the introduction.
(4) Separation between tool code and persistent configuration code is possible. The persistent data management may be revised without effecting the tool code, which is desirable for prototyping.
(5) The persistent information is encoded in a data schema which ranges between an implicit and explicit definition that may be adjusted incrementally. Thus the performance and accuracy of persistent data management may evolve gradually, and the tool designer is not forced to make premature decisions.

5.2.2 Impact of the Requirements
Problem 1: How does the application programmer control the configuration of the persistent data? Since the tool interfaces with a framework a means must be provided to partition the persistent data, transform it to a specific format and transport it to a specified destination. According to requirements 4 & 5 this cannot be encoded in the tool. Therefore a mechanism must provide this functionality in another form. The mechanism should enable the programmer to gradually configure the persistent data.

Problem 2: How does OPI enforce the configuration at run-time? This problem results from the requirements 1, 2 and 3. There is no need for the programmer to write persistent operations using a set of interface functions. Therefore, the dedicated component that performs the run-time data management must be supplied with enough information to perform a partitioning of the object heap. Thus, each object must be associated with a cluster, and with a composite object within that cluster.

5.3 Partitioning Principles

5.3.1 A Vertical and Horizontal Approach to Partitioning
Partitioning is necessary in order to control the distribution of the persistent data in the persistent store. Ecad objects have a strong hierarchical characteristic with regard to aggregation. This implies that from the maze of inter-object pointers (as depicted in figure 5.1
plate A) it is possible to select primary pointers. The primary pointers form a tidy tree structure as depicted in plate B. This structure forms the vertical partition. This tree can be tidied up by capping it with a special system object called Universe (plate C), from which any object becomes reachable through transitive closure. This concept may be upheld by the mechanism of the OPI system. Since any virtual object has a system-pointer (section 4.4), we may define the opposite direction of a system-pointer between two exploded objects to be the primary pointer (plate D).

Figure 5.1. Vertical pruning of objects, an example.

**Definition:**
A partitioning of the object heap implies that each object (instance) within the transitive closure of the universe belongs to one and only one partition element (cluster).

The following assumptions (for virtual objects) hold:
(1) All objects in the heap are unique. (2) All objects belong to the tree. (3) The universe is always a root. (4) Any object has only one parent (or else it is not a tree structure).

**Property:** Given a tree structure, by selecting root objects horizontally along the paths of the tree, a partition is created.

**Proof:**
There is always a partition (3). Suppose that the universe is the only root, then all objects belong to the universe partition and are unique (1,2). Since there is only one partition, the property holds for this case. Suppose there is an additional root along a branch of the tree; This root creates a sub-tree. Any object in the sub-tree is unique (1), and no branches have been
removed or added, thus (2) still applies. This reasoning may be applied iteratively for sub-trees within sub-trees. Let us suppose that an object, say o1, exists in two partition elements, say p1 and p2, with roots r1 and r2 respectively. r1 and r2 must belong to the same path, linking universe to o1 due to (4). If r1 is a sub tree of r2, (or vice-versa) this is a contradiction since a sub-tree creates a new partition element. Thus the only possibility is r1 equals r2, but then p1 equals p2, thus there is only one partition element. Q.E.D.

\[ \bullet = \text{cluster root} \]

Figure 5.2. (Hierarchical) Partitioning of objects, an example.

Any objects along the branch of a root belong to the partition as depicted in figure 5.2, plate A. The marked objects are roots. The height of the horizontal marking determines the scope of the partition. A partition may be sub-partitioned as depicted in plate B. The universe is always considered a root.

We have now established that an object heap partition is upheld by two orthogonal mechanisms; The system-pointers must be set appropriately and key objects must be marked as roots. We call this configuration definition. There are two extremities to approaching the data configuration. One extreme is the static approach, i.e. to hard-wire the configuration in the code. The other is dynamic, i.e. to load the configuration data into the tool at run-time. Various shades lie between these two extremes.

5.3.2 Inter Object Reference Categorization

Definition

A configuration data language (CDL) is a language for defining virtual aspects of application objects, e.g. which objects persist and which are transient, how they are partitioned, etc.

The CDL must allow incomplete definitions. The configuration data always applies to classes. In the degenerate case no configuration information is provided, and the object heap is considered to be one cluster. To create a rough partition one may denote certain classes as roots. It is rough since no pruning has been performed, which implies that the set of possible partitions contains more than one element. Thus, the inter-cluster references are problematic.
To understand the problem, we may perform a categorization of the various interaction possibilities.

![Diagram showing object categorization based on inter-object referencing.]

**Figure 5.3.** Object categorization based on inter-object referencing.

**Definitions**

A **Source Object** is the object in context that contains a pointer to another object.

A **Destination Object** is the object in context that is pointed to by a source object.

A **Source Cluster** is the cluster containing a source object.

A **Destination Cluster** is the cluster containing a destination object.

A **Private (P) Object** is a source object whose destination objects are in its own source cluster and itself is the destination object of source objects within its own source cluster.

(In the context of partitioning) a **Referring (RG) Object** is a source object that might have a destination cluster that is not the source cluster, but itself is the destination object of source objects within its own source cluster.

(In the context of partitioning) a **Referenced (RD) Object** is a destination object that might have a source cluster that is not the destination cluster, but its destination objects are in its own destination cluster.

A **General (G) Object** is a referring and referenced object.

If the source object is private or referenced, or the destination object is referring, then these objects have a **private relation** with each other. Else these objects have a **general relation**.

For implementation reasons, an object is denoted *Unknown* by default, i.e. when its true category is not stated. The worst case situation is then presumed. Figure 5.3 illustrates the various possibilities; An ellipse denotes a cluster boundary. A root object is always referenced or general, because it belongs to its own cluster, but is always referenced by an object outside its own cluster.

The benefit of making these categorizations lies in reducing the amount of information required to perform pruning of the heap, and also enhancing the speed in which the partition can be performed. If an object is referring, it is clear that it belongs to the same cluster as any object referencing it, thus its cluster is known. This is true for a private object, which provides
additional information, namely, that any object it references belongs in the same cluster. Thus, if an object is known to be private, its destination objects may be put in the same cluster, even though they have not been categorized. This holds for a referenced object as well. General is the most useless of cases since apart from notifying the system that it should "look out", it isn't helpful. The categorization is helpful during transformation. (Only objects denoted general or referenced must be left as stubs when a cluster is imploded).

**Property:** The four cases cover all possible inter-cluster interactions. The proof is straight-forward.

It is also possible to perform "gross" categorization, on a per cluster basis:

**Definitions**

A cluster is **Private** if it is a *Private Object* with respect to its parent cluster.
A cluster is **Referring** if it is a *Referring Object* with respect to any destination cluster that does not need to be the parent cluster.
A cluster is **Referenced** if it is a *Referenced Object* with respect to any source cluster that does not need to be the parent cluster.
A cluster is **General** if it is both referenced and referring.

As for object categories, a cluster is denoted *Unknown* if its category is undefined.

---

**Figure 5.4. Hierarchical cluster categorization.**

The references across the cluster boundary of the containing objects make the cluster behave like an ordinary single object, with respect to object interaction. Furthermore the cluster categorization may be extended hierarchically, as illustrated in figure 5.4. The categorization
still applies to the incoming and outgoing references, while considering any sub-clusters as part of the cluster. The usefulness of this cluster information will be demonstrated throughout the chapter and in the next chapters.

**Definition**
A **leaf object** is an object that has no references to other objects.

Two special reference categories are the **private** and **referenced leaf** object. Attributes of leaf objects don’t have to be traversed during partitioning. This reasoning applies to clusters, except that internal references do have to be traversed. Private clusters and **private leaf clusters** are useful for reducing the amount of clusters to be traversed for partitioning (see algorithm **FEW_UNPARTITIONED**, boundary elimination). Note that the Universe cluster is a private leaf cluster.

### 5.3.3 Reference Attribute Categorization
Apart from the object and cluster categorization it is possible to provide more detailed information, by categorizing reference attributes. This is necessary when a referring object refers various objects of different categories, or in different clusters. A reference attribute of an object may be categorized as follows:

#### Definitions
A reference attribute is **composite-private** if it refers to an object within its own composite.
A reference attribute is **cluster-private** if it refers to an object within another composite, but within the same cluster.
A reference attribute is **general** if it may refer to an object in another cluster.
If the source object’s reference attribute to the destination object is composite-private or cluster-private, then these objects have a **composite-private relation** or a **cluster-private relation** with each other. In other cases these objects have a **general relation**.

As in the previous categorizations, a reference attribute is **Unknown** if not categorized, then the worst case is presumed.

This provides the most explicit information and is sometimes necessary if reduction of the partition set to one element is required. Setting the object reference categorization is not enough, because an object needs to be associated not just to a cluster, but also to a composite object. The idea behind the categorization possibilities is that the programmer may select whichever is necessary to get the job done, depending on the stage of the design. During prototyping it is probably easier to focus on the main entities, namely object and cluster categorization.

### 5.3.4 Data Configuration Concerning the Database
Designating certain roots as **database-objects (DO)** defines the scope of a database package as explained in section 4.4.4. A DO root implies that all sub-clusters will be transported to the
same database object, excluding sub-clusters that themselves are DO's.

\[
\text{○} = \text{cluster root} \\
\text{□} = \text{DO cluster}
\]

![Diagram of database object clustering](image)

Figure 5.5. (Hierarchical) Database object clustering, an example.

**Definition**

An object is **Floating** if there is not enough information to associate it to a definite cluster.

During the final partitioning process, Floating objects are attached to what the system considers the suitable cluster. Floating objects are not desirable if the references cross DO clusters, since the placement of such objects might be correlated with the DBMS operations on metadata. For example, in the NELSIS framework, it is possible to remove a package at the database level. Such an operation can cause data corruption.

**Example**

In figure 5.5 cluster A includes a DO cluster B which will be assigned to a separate database package, while in the case of cluster C, all sub-clusters are assigned to the same package.

**5.3.5 Conceptual Partitioning Schema**

In figure 5.6 the implicit information schema has been extended with the partitioning information.
A specialization virtual-user-class has been added, since additional attributes are added. reference-type: This entity implies reference categorization according to class type. It may have the values private, referenced, referring, general and unknown. Unknown is necessary in two cases: If the user doesn’t define any reference information about a class type, or when a class type may be several reference-types, depending on its hierarchical placement, and they are not separately defined per hierarchical placement.

Aggregate-type: This entity enables to specify the type of aggregate of the class type, namely, object, composite object, cluster, db-cluster, and unknown. Unknown is used as in the previous case. Apart from an object categorization, a cluster path categorization may be defined using the following relation.

Hierarchy: This defines an aggregate relation such that the parent (denoted P) is the container class and the child (denoted C) is the contained class. aggregate-type and reference-type apply in this case always to the child. This follows from the fact that the top of the hierarchy is universe which is considered to have reference-type private and aggregate-type cluster.

Apart from these additions, another specialization virtual-attribute has been added, since an attribute is added.

Relation-type: This entity may have the values general, composite-private, cluster-private and unknown. It enables fine-tuned partitioning, since e.g. two different attributes belonging to the same class type, could refer to different class types, while one could be of type general and the other of type composite-private. Several assertions hold for the relation-type of attribute, and the hierarchy relations.
5.4 Run-time Data Configuration Enforcement

Virtual objects are not necessarily associated with a cluster since this implies setting the system-pointer to an explicit object. In principle, this can be hard-wired in the code (method 1 of appendix 4.1.3), which is by far the most efficient method. However, during tool development, where modifications are frequent this is less relevant. It is useful to first tune the tool and finally perform such hard-wiring if performance is an issue. If the system-pointer is not hard-wired, then the partitioning is accomplished by utilizing the available configuration information in order to form clusters, which must be done when data in the object heap needs to be transported, e.g. saved. Three algorithms are used for dealing with the various possibilities. One "algorithm" is used when there is no configuration information; This means there is not much to partition, and so "algorithm" is a big word for it. Another algorithm is used when there are many objects to partition or the configuration information is not detailed. Finally, one algorithm is used when only a few objects need partitioning, and the configuration information is detailed. The efficiency of the algorithms is important because setting the system-pointer as quickly as possible is a run-time performance problem.

5.4.1 Infra-structure for the Partitioning Algorithms

The partitioning algorithms work on the generic infra-structure of the system, which we shall first briefly describe. A number of special purpose components play a role in the partitioning as depicted in figure 5.7:

![Diagram](image)

**Figure 5.7. Partitioning infrastructure.**

Universe cluster is the top level cluster in the process.

**Definition**

The Unpartitioned cluster is a cluster that contains all objects that have not been partitioned (i.e. do not belong to a partitioned cluster).
The Partitioned cluster represents all other clusters.

The Uncomplete list is an intermediate stage each unpartitioned object has to go through, before it ends up partitioned. This intermediate stage is necessary to be able to handle newly partitioned objects separately.

5.4.2 Algorithm 1: NO\_CONFIGURATION\_DATA
Absence of configuration data. Since there is no information on how to partition, all objects are by default clustered in the Unpartitioned cluster. The Unpartitioned cluster is managed as a normal cluster, and will be transported to a pre-defined location. When an opilnii is invoked, the Unpartitioned will be automatically exploded. If the programmer performed hard-wiring of user-clusters using interface functions, then the Universe cluster and hard-wired user-clusters are traversed, and all unpartitioned objects referenced by a hard-wired or previously partitioned object in a cluster are partitioned in that particular cluster. All other unpartitioned objects remain in the Unpartitioned cluster and are treated as described earlier.

5.4.3 Algorithm 2: MANY\_UNPARTITIONED\_OBJECTS
Presence of configuration data and a large number of unpartitioned objects.

![Diagram](image)

**Figure 5.8. Many unpartitioned partitioning problem.**

If the number of unpartitioned objects is large, it can be better to traverse the entire heap beginning from the universe cluster, since the algorithm is faster for this case. We apply the following traversal method:

All objects partitioned to the universe are traversed. Each partitioned object is requested to partition each referenced object. If the referenced object is not partitioned, then the partition manager consults the configuration data to check whether the referenced object has a composite-private relation (if defined, else private relation) with the partitioned object. If it does, the referenced object’s SP (system pointer) is re-set from the unpartitioned root to the partitioned object and it is appended to the partitioned object’s cluster. A newly partitioned object is put in the uncomplete list for traversal of this object too. If the partitioned object is not definitely the owner, then the referenced object remains in the unpartitioned cluster and the
referring partitioned object is stored in a list of the referenced object. Once all objects in the universe cluster have been traversed (including appended ones), the clusters of the universe cluster are iteratively traversed like the universe cluster. This causes each cluster to perform the algorithm described above. After each unpartitioned object that is partitioned, the unpartitioned cluster is checked. If it contains no more objects, then the partition is complete. Otherwise, the partitioning process percolates down to the child clusters. If the whole heap has been traversed, and the unpartitioned cluster is not empty, then the list of referring objects of each unpartitioned object is checked. If there is more than one candidate, a choice is made. If there is no candidate, it implies that the object has not been reached from the universe or an error has occurred. If there is no error, it remains clustered in the unpartitioned cluster and is managed as in the previous algorithm.

5.4.4 Algorithm 3: FEW_UNPARTITIONED_OBJECTS
Presence of configuration data and a small number of unpartitioned objects. In this case the overhead of traversing the whole heap is considered too high and an attempt is made to perform the partitioning from the direction of the unpartitioned objects. This means that the references of the unpartitioned object are traversed.

![Diagram](image)

**Figure 5.9. Few unpartitioned partitioning problem.**

A three phase algorithm is employed to find the cluster of an unpartitioned object.

If the cluster isn’t found, then the partitioning is unresolved, and the unpartitioned object remains clustered in the unpartitioned cluster. A notification of this event is recorded if required, or the whole heap is traversed as in the previous algorithm. We now present a more detailed discussion of the various phases.
The direct phase
In the direct phase immediate information is used to find the cluster of the unpartitioned object. The unpartitioned object is examined for references to partitioned objects with a private relation; If such a relation is found, the unpartitioned object is clustered in the cluster of the partitioned object. Only one cluster needs to be traversed in the traversal phase to find the composite object.

The elimination phase
In the elimination phase an attempt is made to eliminate all cluster candidates for traversal that really do not have to be traversed. Additional configuration data is necessary for the elimination phase; Each object class can have one or more containing cluster class definitions, to define the class of the cluster instance in which an object is clustered. The elimination of nonrelevant cluster instances can be divided into three categories, that utilize the properties of objects and clusters described in sections 5.3.2 and 5.3.3 respectively. categorization:

1. The class category eliminates all cluster instances not belonging to the containing cluster class of the unpartitioned object. This category eliminates "horizontally".

2. The path category is only necessary, when the unpartitioned object may be contained in more than one cluster class. In that case the path category eliminates all the containing cluster classes different than the containing cluster class of an unpartitioned referenced object with a private relation. (i.e. all cluster classes, with a different path than the containing cluster class of an unpartitioned referenced object, are eliminated.)
3. The **boundary** category traverses all parents of each of the partitioned referenced objects (with a general relation), until it finds a private cluster instance. All candidate cluster instances need the same base path as this private cluster instance. A private leaf cluster can be added to this base path (so eliminating even more clusters). This category eliminates "vertically".

Note that a boundary elimination is not possible for leaf objects. Also note that in path elimination a path of classes is meant, and in boundary elimination a path of instances is meant. The eliminations are performed in the following order: class, path (if necessary) and finally boundary. An alternative order would be: boundary, class and path. The first order is selected because a partitioned cluster is put in a list of its classtype. So if the containing cluster class of an unpartitioned object is known, then a class elimination is already performed. The order is a matter of how the partitioning manager is implemented.

The final result of all elimination operations is a set of cluster instances. These are passed to the traversal phase. When no cluster instance passes the eliminations, we have an unresolved reference.

**The traversal phase**

During this phase the elimination output is traversed in order to locate the cluster and composite object of the unpartitioned object. If the cluster is found in the direct phase then this cluster is traversed for the composite object. When a cluster is traversed, all its objects are traversed for references to unpartitioned objects. If an object has a composite-private relation (if defined, else private relation) with the unpartitioned object then the unpartitioned object is clustered with this object. In the case of several cluster candidates then the first candidate is selected, and if there are several composite object candidates (i.e. there is no composite-private definition) then the first is selected too.

In the appendix of this chapter a detailed description of an example implementation of the FEW_UNPARTITIONED algorithm can be found.

### 5.4.5 Partitioning Anticipation

A possibility to speed-up the partitioning is to anticipate who the rightful owner of an object is. When studying the CFI [17] interface functions it becomes quite clear that objects are usually created by their owner. Suppose an object knew who the sender of a message was, it could register the sender if desired. For example, upon creation, it could register the creator, which is called the **potential owner**. The ownership may not always be correct, thus a hit rate may be measured. The partitioning begins by first picking the potential owner as the source object. If the ownership proves incorrect then the loss is minimal, however if correct, a speed-up can be gained. There are three possibilities for extracting the **potential-owner** information:

1. The sender's ID is passed as a standard argument.

   This may be standardized by agreeing that the constructor method always passes the
sender. The disadvantage is that the programmer is required to uphold the mechanism in the code. However, it is a simple procedure. For example:

```c
    cell = lib->cfidrCreateCell("cell1");
```

The implementation of `cfidrCreateCell` contains the line:

```c
    cell = new cfidrCell("cell1")
```

can be called with an additional variable, the creator `lib` (this):

```c
    cell = new cfidrCreateCell("cell1",this);
```

and `cell` registers `lib` as primary owner candidate in its temporal storage space.

2. The compiler of the language supports such a mechanism.
   This is not the case for the standard object-oriented languages.

3. An ancillary mechanism is built.
   This implies that in general any message must trigger a registration of the instance involved. When the constructor operator is invoked (a new object is created) the instance currently in the register becomes the potential owner. The overhead may further be minimized by reducing the triggering to selected methods, by defining this in the configuration data.

---

**Appendix 5.1 Local Universe Variant for Virtual Object Addressing**

The addressing mechanism (section 4.6.1) may be enhanced by looking at the cluster categorizations. If a cluster is private or referring then by definition it cannot contain objects that are referenced from outside the parent cluster of the cluster. Therefore, the concept of universe may be impacted to be the root of the parent cluster, i.e. a local universe concept. If a cluster is a private leaf cluster then the cluster itself is a local universe. Within the local universe, the same addressing mechanism applies. This reduces the length of logical IDs stored by imploding referring objects and shortens the computation steps required to locate an object. This is favorable for slot recycling, since it can be done on a per local universe basis, which is more practical than loading in the complete universe.

---

**Appendix 5.2. Implementation Aspects**

The schema of figure 5.6 is conceptual. The implementation was dictated by available software and manpower resources. A syntax was devised, that could be read by a simple parser. A less specific data-schema has been used, by utilizing the general services of the NELSIS framework.
Appendix 5.2.1. The configuration data language (CDL) syntax.
The CDL enables the definition of configuration objects. Each configuration object represents a
cluster or a class and has a set of attributes that enable accurate definition of certain
configuration aspects such as transformation and transportation, whether it is a root object, the
type, selection of persistent attributes etc.

definition list  ::=  <type definition> I <definition list>
                    <type definition>
type definition  ::=  APPLICATION <name> I
                       CLASS <classname> [DOCLUSTER I
                       CLUSTER [<clustertype>]
                       CHILD OF <classname>]
                       [RELATION <ONEIMANY>
                       TO MANY] I
                       TRANSPORT <name> I
                       TRANSFORM <name> I
                       ATTRIBUTE <name>
                       PERSISTENTVOLATILE
                       [POINTER [[TO <classname>]] I
                       [ENTITY <entityname>]]
                       [RELATION <ZEROIONEIMANY>
                       TO <ONEIMANY>]

clustertype  ::=  PRIVATE I REFERENCED I
                  REFERRING I GENERAL
entitydef  ::=  ENTITY <entityname> = <classlist>
classlist  ::=  <classname> I <classlist>
                <classname>
entityname  ::=  <name>
classname  ::=  <name>
name  ::=  [a-zA-Z][a-zA-Z0-9]+

Appendix 5.3 The Configuration Data Schema

The data schema shown in figure 5.11 is derived from the NELSIS data schema [10] and is
defined according to the OTO-D variant used in the NELSIS design system. The data types to the
right of the separating line are used by the framework for general management purposes. In
this context the DesignObject represents a configuration object which has attributes like Type
and Module pertaining to its behavior. A Module is a compound data type that consists of an
Applic, ViewType and Name attribute.
Figure 5.11. The configuration manager data schema mapped to NELSIS.

Applic denotes a specific application in which the object is used (one or more tools) and Viewtype distinguishes between clusters and classes. Type denotes the status of the class, e.g. Private, General, etc. The configuration objects may be related via the Hierarchy and Equivalence relations. We recognize two types of hierarchy: cluster and class, and two types of equivalence relations: transform and transport.

A set of default methods is defined for transport and transform operations, which is normally sufficient, but in special cases may be overridden. In this case the configuration data may specify (a reference to) the configuration code, written by the tool programmer.

The configuration management data is managed by a NELSIS framework project called the CM pilot which is configured according to a data schema specific to the CM. Thus the CM acquires all the framework related data management facilities.

Appendix 5.3.2. The schema types.

View Types
There are two categories of View-type: Class and Cluster. Cluster contains configuration objects of type cluster and Class contains configuration objects of type class.

Hierarchy
In the top left corner is Hierarchy. It is used to capture the hierarchical relation between classes as defined by the inheritance relation of the object oriented language (thus only used in Viewtype Class).
Equivalence
In the top center is **Equivalence**. **Equivalence** is used to capture the following relations:

1. The recursive clustering is captured by equivalence (because cyclic relations are possible). The **Original**-**do** is the cluster closest to **Universe**. The Info field contains the value 'PATH'.
2. The relation between a Class that is a Cluster. This relation crosses the **ViewType** boundary. Info field contains the value 'PATH'.
3. Equivalence relation in **ViewType** Class between class/transform. Info field contains the value 'TRANSFORM'. The transform function is represented by a Class (logical).
4. Equivalence relation in **ViewType** Cluster between cluster/transport. Info field contains the value 'TRANSFORM'. The transport function is represented by a Cluster.

**OPI-object (DesignObject)**

1. **Type:**
   - In **ViewType** Class the Type denotes REFERENCED, In **ViewType** Cluster the Type denotes REFERENCED

**Module**

1. **Applic:** defines the application domain of the object in context.
2. **Name:**
   - Is the name of the class/cluster.
3. **LastVersion:**
   - Is used by the **NELSIS** framework.

**Vnumber & Vstatus**
The **Vnumber** (version number) attribute enables defining many configuration-objects using the same class (module) per application. The meaningful **Vstatus** (version status) values are *Actual, Working & Backup*.

**Transaction**
Transaction has the same meaning as in the standard **NELSIS** data-schema, i.e. keeps a log of tool invocations, and the related configuration-objects manipulated by the tool.
Appendix 5.3.3 The CM tools.

The following tools may be invoked by the application programmer in order to manipulate the CM pilot.

[1] opiCMUI (CM user interface) is a graphical user interface for invoking and monitoring CM tools.

[2] opiMkCO (make a configuration object) - parses the definition of a configuration-object written in the configuration language (OPICL).

[3] opiDefCO (define a configuration object) - extracts a configuration object definition from the CM pilot, or provides a template definition.

[4] opiDefCL (define an item of an configuration object) - retrieves an item from the CM pilot or provides a template. Items are user defined textual data which is a subset of an configuration objects' design data.

[5] opiRmCO (remove configuration object) - invokes the NELSIS Framework tool rmCell.

[6] opiGenCC (generate Configuration Code) - opiGenCC generates initialized structures containing configuration data that may be dynamically created in a tool’s process space. These structures are tailored to enable quick access (using pointers) to the configuration data. A library of functions provides general querying and modification of this
configuration data heap [54].

[7] opiInstallCT (install Configuration Test) - prepare ANT system for testing the specified configuration. The test phase is recommended but not obligatory.

[8] opiRetrieveCO (retrieve Configuration Object) - retrieve older test results of a configuration object.

[9] opiMkCM (make CM) - create a Configuration Manager Pilot. This invokes a NELSI tool for creating a project (configured to the CM data-schema).

Appendix 5.4 The FEW_UNPARTITIONED Algorithm

The algorithm is implemented in C++, and is performed by the partitioning manager.

For this implementation a few assumptions are made:

1. Each unpartitioned object class has at least one containing cluster class definition in the Configuration Data Heap. It is possible for this algorithm to work for unpartitioned objects without this definition, when the list of unpartitioned objects is sorted in advance. If unpartitioned objects with the highest probability of finding the smallest amount of containing clusters are put in the list first, then it may be possible that unpartitioned objects with little configuration data are "accidentally" partitioned without needing to traverse the whole heap (like MANY_UNPARTITIONED).

2. All partitioned cluster instances are inserted in the list of their cluster class definition.

3. Each unpartitioned object has at least one referring object with a private relation. Otherwise no object to attach to will be found. This means NO floating objects.

The following types (classes) are used:

 opiList is a general purpose list object.
 opiSystem is the base class for handling the user objects. (Users should use opiVirtual.)
 opiCluster is the cluster object attached to root objects.
 opiConfigData is the configuration data heap manager object.
 opiReferencedObjects is defined below.
typedef struct _opiReferencedObjects
{
    opiSystem * ParPrvRel; // Partitioned Referenced Object with
    // Private Relation
    opiSystem * ParGenRel; // Partitioned Referenced Object with
    // General Relation
    opiSystem * UnpPrvRel; // Unpartitioned Referenced Object with
    // Private Relation
}opiReferencedObjects

opiResult opiPartition:: opiPartitionFewObjects(int RunNumber,
    opiCluster * UnpartitionedCluster,
    opiCluster * UniverseCluster,
    opiConfigData * ConfigDataManager)

// RunNumber: Each partitioning run has a number, and each cluster has the
// number of the partitioning run during which the cluster was last traversed.
// Each time a cluster is traversed, this cluster gets the current RunNumber to
// prevent this cluster to be traversed more than once during a single
// partitioning run.
// UnpartitionedCluster: the cluster containing the objects which need
// partitioning.
{
    opiList * UnpObjLst; // Unpartitioned Objects List
    opiSystem * UnpObj; // Unpartitioned Object
    opiReferencedObjects * RefObj; // Struct containing the three Referenced Object
    // sorts
    opiSystem * ParRefObjPrvRel; // Partitioned Referenced Object with
    // Private Relation
    opiSystem * ParRefObjGenRel; // Partitioned Referenced Object with
    // General Relation
    opiSystem * UnpRefObjPrvRel; // Unpartitioned Referenced Object with
    // Private Relation
    opiList * ObjectClassLst; // Object Classes List (ConfigDataHeap)
    opiList * ConClusInstLst; // Containing Cluster Instances List (ConfigDataHeap)
    opiList * BoundPathLst; // Boundary Cluster Path List (of cluster instances)
    opiCluster * BoundaryCluster;
    opiCluster * Cluster;

    UnpObjLst = UnpartitionedCluster-> opiMyObjectsList();
    // Get list of unpartitioned objects from the Unpartitioned Cluster.
// It is in this case not necessary to start in a special order,
// so a random object may be taken to start with.
// (For further optimization the list may be sorted.)
while(UnpObjLst-> opiNumberOfElements() > 0)
{
    UnpObj = ( opiSystem *) UnpObjLst-> opiRemoveElement();
    // The element currently pointed at is read and after that removed
    // from the list. The next element will be pointed at next. etc.
    if((UnpObj-> opiNotPartitioned()) &&
        // An Unpartitioned Object from the list may already be partitioned in
        // the mean time, and therefore skipped.
        ((RefObs = UnpObj-> opiMyRefObjOfSort() != NULL))
        // {PP,PG,UP) = opiMyReferencedObjectsOfSort()
        // Traverse the Unpartitioned Object to find:
        // - a Partitioned Referenced Object with a Private Relation (PP)
        // - a Partitioned Referenced Object with a General Relation (PG)
        // - an Unpartitioned Referenced Object with a Private Relation (UP)
        // If opiMyRefObjOfSort returns NULL (It was unable to determine
        // PP, PG and UP, because of no config data or an error.),
        // then this UnpObj is skipped.
        
        { // Try Direct phase first.
            if((ParRefObjPrvRel = RefObs-> ParPrvRel) == NULL)
            {
                // Direct phase not possible: go through Elimination phases.
                // Class Elimination phase:
                ObjectClassLst = UnpObj-> opiMyObjectClasses();
                // Get list of ObjectClasses of the Unpartitioned Object class
                // from the Configuration Data Heap.
                if((ObjectClassLst-> opiNumberOfElements() > 1) &&
                    ((UnpRefObjPrvRel = RefObs-> UnpPrvRel) != NULL))
                    // Is Path Elimination necessary, and possible ?
                    {
                        // Path Elimination phase:
                        ObjectClassLst = UnpRefObjPrvRel-> opiMyObjectClasses();
                        // Get a list containing only one ObjectClass of the
                        // Unpartitioned Referenced Object with a Private Relation.
                        
                        ConClusInstLst = ConfigDataManager->
                        opiGetConClusInstances(ObjectClassLst);
                        // Get list of Containing Cluster Instances through the
// ObjectClassLst. In the ObjectClass(es) of an Object, the
// Containing ClusterClass(es) of that object is defined. From that
// (those) ClusterClass(es) the list(s) of cluster instances is
// copied. (For each cluster class a list of cluster instances is
// created.) In most cases there will only be a list of cluster
// instances of one class.
if((ParRefObjGenRel = RefObjs->ParGenRel) != NULL)
{
  // Boundary Elimination phase:
  BoundCluster = ParRefObjGenRel->opiMyBoundaryCluster();
  BoundPathLst = BoundCluster->opiMyClusterPath();
  // Get cluster instances Path of the Boundary Cluster of the
  // Partitioned Referenced Object with General relation.
  // The boundary cluster is the parent of a containing private
  // cluster or a containing private leaf cluster.
}
else
{
  // No Boundary Elimination possible:
  BoundPathLst = UniverseCluster->opiMyClusterPath();
  // In all other cases the Universe Cluster is the Boundary
  // Cluster.
}
while(ConClusInstLst->opiNumberOfElements() > 0)
{
  Cluster = (opiCluster *) ConClusInstLst->opiRemoveElement();
  if(Cluster->opiEqualBasePath(BoundPathLst))
    // Is the path of the Cluster equal to the path of the
    // Boundary Cluster, or are the first cluster instances of
    // the Cluster path equal to the cluster instances of the
    // whole path of the Boundary Cluster ?
  {
    Cluster->opiPartitionReferencedObjects(RunNumber);
    // All previous and new partitioned objects in the cluster
    // are traversed and examined for unpartitioned referenced
    // objects with a private relation.
    // Newly partitioned clusters will be traversed too !
    // Note that more unpartitioned objects than the
    // Unpartitioned Object may be partitioned.
    // If the internal RunNumber of the Cluster is equal to the
    // RunNumber (The cluster has already been traversed.) then
// the Cluster will not be traversed again.
}
}
else
{
    // Direct Phase found a single cluster:
    Cluster = ParRefObjPrvRel->opiMyCluster();
    Cluster->opiPartitionReferencedObjects(RunNumber);
    // Traverse the owner cluster of the Partitioned Referenced Object
    // with Private Relation.
}
}
}

if(UnpartitionedCluster-> opiNumberOfObjects() == 1)
    // The Unpartitioned Root Object remains in the Unpartitioned Cluster.
{
    return(OPI_SUCCESS);
    // Ready.
}
else
{
    return(OPI_FAILED);
    // Error, or insufficient configuration information.
}
Chapter 6

Fine Grain Data Management Utilizing a VOIS

6.1 Chapter Overview

A problem with current data management systems is the overhead incurred in performing the design data management at run-time. This is one reason why many tool programmers choose to use the ad-hoc (but efficient) method of wiring the semantics they consider important into the tool code. In this chapter we present solutions to several problems related to fine-grain data management within the context of a VLSI tool-set incorporated in a framework. The following problems must be solved:

How to define an information model that can cater to the needs of the tool programmer and the environment builder at the same time?

The information is often too specific or generic. By supporting a model that may be defined in a unified textual schema this problem is solved.

How to enable a trade-off between (static and dynamic) design consistency and performance?

The trade-off problem is solved by enabling incremental addition of data to specific parts. Due to the different characteristics, the management of the large-grained data is handled differently from the fine-grained data to optimize the performance.

The structure of this chapter is as follows: Section 6.2 presents aspects of fine-grained management. In section 6.3 we discuss modeling. Section 6.4 we divide the interface mismatch problem into three aspects and section 6.5 presents our approach to supporting the tool programmer’s and the environment builder’s modeling needs. In section 6.6 we discuss the incorporation of the design manager on top of the virtual object system to enable incremental addition of management. Section 6.7 shows the architecture of the system, and section 6.8 illustrates how to extend the virtual system. Section 6.9 is a case study illustrating how the concepts are applied to VLSI data management. We conclude in section 6.10. (This chapter has been compiled from [55]).
6.2 ECAD Environments

Commonly, ECAD tools are incorporated into a framework. The classical approach is to make no distinction between large-grained and fine-grained design data. The performance problem, however, has led several framework researchers (in the field of ULSI) to the conclusion that the framework must provide design management services only for the large-grained data. Systems coupled to frameworks not adhering to this concept pay a high run-time penalty. The end-user demand for fine-grained design management has forced researchers to provide a solution. Several emerging state-of-the-art systems are therefore based on a dual design-management system, one for large-grained and one for fine-grained which are linked to form a coherent entity. Confusion arises as to what exactly is the framework i.e. does it encompass the fine-grained management as well? We don’t attempt to answer this question, but in our case, we are using an established framework [10] that provides large-grained management, on top of which an experimental fine-grained system is built. Therefore, in this context, the fine-grained management is ancillary to the framework.

Definition

Environment building is the act of incorporating a set of tools in a framework with the minimal requirement being the sharing of essential design data.

Environment building requires a common data representation for the use of the tools and the framework. An interface system provides a set of intermediate operations, some of which are controllable by the tool programmer and/or the environment builder.

6.3 Modeling Concepts

The semantics and mechanisms required for CAD modeling are being studied and implemented in various projects, such as [56, 17, 45, 57]. A common denominator of the emerging models is the object-oriented modeling approach to design data, the basis of which is encapsulation and inheritance. A formal means of defining the semantics is called a data model. A particular combination of classes and their inter-relation is encapsulated in a data schema. A data definition language (DDL) enables defining (textually and graphically) a schema according to a particular data model. A data manipulation language (DML) enables manipulation and querying of actual objects in a functional model.

In OPI, the level of semantic modeling and consistency may be controlled by the environment builder and the tool programmer. The mechanism is built of two sub-systems, a virtual object manager, that performs partitioning of the design data into clusters based on the hierarchical structure of the design data and a fined-grained design manager which is incorporated on top.
6.4 Considerations And Problems

6.4.1 Interface Mismatch Problems
To reduce their complexity, large design systems are constructed of functional software layers, the basic three being the operating system, the framework and the tools. This layered structuring leads to the problem of interface mismatch between interconnecting layers (because the different components are usually supplied by different vendors). Since our work focuses on the interfacing between the framework and the tools we wish to study various aspects listed below:

The Data Model Aspect:
The data-model requirements for the actual tool implementation are different from those for the common data model necessary for representing data to the framework. Part of a tool’s data is volatile and has no bearing on the persistent data managed by the framework. The design data varies between tools. The large part of the behavioral semantics are not relevant for the framework. The structure of the data is different and usually far more detailed in the tool. The trade-off between the extent of the semantic modeling and the run-time performance needs to be controllable.

The Data Granularity Aspect:
The huge quantity of data forces different management of fine-grain and large-grained design data. Metadata describes the relations between design data. In the case of large-grained data, the metadata comprises only a small percentage of the design data. In the case of fine-grained design data, to a large extent, the metadata is the data therefore managing a separate set of data structures or tables for this data is a duplication.

The Programming Aspect:
As explained in chapter 2, the task of tool integration is most challenging at the programming level. The programmer must constantly be aware of differences in manipulating persistent design objects and s/he is usually limited to using a predefined set of manipulation functions in the form of a procedural interface. A virtual system, such as OPI, alleviates this problem.

6.5 Modeling of Fine-Grained Design Objects

6.5.1 Modeling Facets
Definitions
There are two facets to fine-grain modeling of design objects. The first is the internal model which supports the tool-programming, and the second is the external model which supports the environment building.
If the internal model is supported then the level of detail for the environment builder is too large. In addition, in the case that the external model is supported then the tool programmer is faced with the problem of mapping the tool to the external schema.

The internal model must support the tool programmer’s environment. In our approach the virtual object management system plays a central role, therefore we presume that the programming environment is an object oriented language (supported by OPI). We shall extend the OIS to a model which we shall call OIS+. This model is, like OIS, in essence the Express-G data model. The graphical representation however, has been adapted to correspond closely with OIS. The model is simple and enables modeling at a level suitable to support the tool programmer’s requirements since the graphical data schema maps easily to data-structures and pointers. Figure 6.1 is an example of the graphical representation. A rectangle represents a class. An arrow represents a (directed) relation. An arrow leaving the corner of a rectangle (A) with its head at the corner of another rectangle (B) implies that the class A inherits B. If the arrow is bidirectional, it implies that class B is aware that it has a specialization A. The specialization is always the upper class. An arrow leaving the middle of a rectangle (B) and ending in the middle of another rectangle (C) implies that class B has a reference to class C e.g. via a C-pointer.

A is_a B  
B relates to C  
B relates to D  
D relates to B  

Additional Data:  
B-C: 1-1  
B-D: List [0:n]  
D-B: 1-1  

Figure 6.1. Class attribute schema example.

An arrow from the edge of one class to another implies that one or more (non-inheritance) relations exist between the two classes, and the relation is known to the pointer class. The programmer can specify the type of relation, number of relations or the cardinality in the graphical schema itself or separately in a textual representation which is the syntax of the object-oriented programming language itself. We shall not specify the type of relation, number of relations or the cardinality in the schema, to achieve a good resemblance with the external schema.

In our approach the CDL [2] (defined in appendix 5.2.1) is extended. Since the essential data is defined in the code, the data in the CDL may be added incrementally to the attributes and classes of ones choice, and these alone will be checked for consistency. The check is (by default) static, i.e. when a snap-shot of the object heap is performed. Thus, this has no bearing on the dynamic performance. For the external model we use OIS as explained in section 4.2.
6.5.2 Bridging the Gap

Our strategy is to work top-down from OIS towards C++. The persistence characteristic of OIS instances is supported by the fact that all types inherit the virtual object class Virtual as depicted in figure 6.5. The other differences may be overcome implementing the missing integrity constraints of OIS in the form of methods defined in two classes, OISType and OISManager (figure 6.5). All classes corresponding to types in the data-schema may be defined as direct or distant subclasses of Type. Thus the application classes inherit additional semantic behavior. The OISManager has a view over all the classes and instances and therefore can derive the relational aspects between objects.

The integrity of the data is checked only after a snap-shot of the heap has been performed, due to the extensive checking involved. The additional data needed per class is added in the CDL, i.e. composite aggregations, etc. These will be explained in the case study (section 9). Integrity checking is only performed on the classes that have additional OIS data, thus enabling a trade-off with performance, in this case the time it takes to update the database.

6.6 Incorporation in a VOIS

6.6.1 Interaction between Components, Players and Schemas

We first clarify the interaction between the components, players and schemas as depicted in figure 6.2. Players are persons who can assume various roles. We define a component as a software module within the environment and a player as a person that may manipulate certain components.

![Diagram of OIS schema of the relation between components and players.](image-url)

An environment consists of a framework, an interface, tools and an environment builder who
manages the environment. The framework is managed by the frame manager, and has an OIS schema for the large-grained data. The interface has an OIS and a OIS+ schema, which are commonly shared by the tools, and is defined by the environment builder. A tool is managed by a tool programmer. Each tool has its specialized OIS+ schema which consists of extensions to the OIS+ schema of the interface. The environment builder is the central player, and communicates requirements for the framework’s large-grained schema to the framework manager, and gathers information from the tool programmers pertaining to tool’s requirements for persistent storage and integrity requirements.

6.7 Overview of the Architecture

The architecture of the system has 4 management layers as shown in figure 6.3. The arrows depict interaction of the tool and sub-modules. The UNIX system provides the base, on top rests the framework that performs the large grained data and design management. The virtual object system provides simple fine-grained data management. The fine grained design manager is incorporated on top of the virtual system.

![Diagram of system architecture]

Figure 6.3. System architecture.

Only the VOIS has an internal (OIS+) and external (OIS) schema, which is common to all the tools. Each tool may have an additional internal schema to add tool-specific information. The tools that belong to an environment may vary in their level of interaction with the system; some might be integrated directly on top of the framework, others might use the virtual system, and others might in addition use the fine-grained design management facilities as explained in section 1.3.
During invocation of a tool, the fine grain data manager contains high-level design information as well, which is initially obtained from the framework's external schema. Thus within the tool's workspace there is no separation between large and fine grained metadata. The tool's metadata reflects only a small portion of the total metadata, i.e. of the design objects loaded into the virtual system. During the tool invocation the metadata of the framework will be modified to reflect any high level changes caused by the tool (see figure 6.4). The large grain objects are known to the framework since they are registered in the framework's metadata. After an object is checked out (loaded into the tool's working memory) the fine grain objects become visible to the tool. The tool has metadata of its own concerning the fine grain objects. This fine grain metadata is managed by the fine grain design object manager and only affects the persistent objects.

![Figure 6.4. Connection between fine and large grained data.](image)

### 6.8 Extending the Virtual System

In a system like OPI data is loaded on demand which implies that the management of (fine grain) design objects must be performed piecemeal. The demand load facilities are already available and implemented as classes. Extending the system by implementing data management in classes implies that demand loading and data management are coupled. Thus, only the necessary data is present and the data management is transparent to the user. In figure 6.5 a partial view of the class architecture of the system is shown. There are 3 class categories; OPI classes (System and Virtual as discussed in chapter 4), the design manager classes which form the design management system (OISManager, Equivalence and OIS Type) and the application classes which are user defined extensions to the former two categories (Userclasses and Transactions).
The configuration data manager (CDM) performs the virtual functions using the configuration data heap (CDH, appendix 5.2). The relations between the CDH and the design objects form the basis for the semantic OIS design management.

![Diagram](image)

Figure 6.5. Coupling the virtual object and the design management systems.

6.8.1 The Fine Grain Design Object Manager

The configuration data manager has been explained in chapters 4 and 5. The design manager consists of two main classes that interact closely with the virtual system and the user defined classes. The main function of the design manager is to perform semantic and syntactic checks which are needed to uphold the consistency of the design data. The design manager contains a query module that enables it to check semantic and syntactic aspects of the design data. To perform these tasks, the design manager retrieves information from four different sources:

Tool OIS+ schema:

This schema is an abstract information model that is hard-wired into the object classes in the object heap and is derived from the data contained in both the CDH and the external representation of the environment.

OIS schema:

This schema is a static pool of data that together with the data contained in the CDH provides enough information to perform queries on the design data. From the information in the CDH and the tool OIS+ schema, semantic checks may be performed on the design data.

The CDH:

The CDH is a partially static, partially dynamic data pool that contains the configuration of the design data. Since the configuration may be performed piecemeal, the design manager may only perform after a partitioning has been performed on the object heap and all the relations are known.
The design data:
The design data is located in the object heap of the virtual system. The objects have relations according to the tool ois+ schema.

6.9 A VLSI Interface Case Study

In this study we shall discuss the application of the design manager to the CFI Design Representation Model and Programming Interface [17]. This interface specification is utilized in several chapters, therefore the level of detail here is slightly higher than required for this chapter.

6.9.1 The CFI Design Representation Information Model

CFI (CAD Framework Initiative) is an international project intended to solve the problems related to tool interfacing. Several major CAD software vendors and work-station suppliers have key positions in this project. CFI is a success in the sense that a clear statement has been made regarding tool interfacing, namely, it is a problem that is costing vendors and consumers significant financial resources.

CFI's approach to tool DBMS interfacing is conceptually simple in terms of software technology: define a large library of well defined functions that are called through a C procedural interface. In terms of architecture, there are similarities with a VOIS in the sense that fine grained data are considered objects and the persistency aspects are largely supported by the library. CFI does not say how this should be implemented, since its primary focus is standardizing the Electronic CAD (ECAD) library definitions.

Figure 6.6 shows the net-list data schema based on the Express-G data model. The notation is as follows: A line between rectangles represents a relationship. The circle denotes the relationship direction. A shaded rectangle represents an object type. An unshaded rectangle represents an object type that has a relation to one of many object types, for example, a PortOwner may have a relation with either a View or a PortBundle. A thick line represents an inheritance relationship. L[0:#] denotes a list of zero or more objects with a given relationship. Each type of object in the schema has a set of well defined methods that enable manipulation of instances of the type. All objects inherit a BaseObject that has generic methods e.g.:

```cpp
ObjectTypeT BaseObject::GetObjectType ()
```

defines (in C++) a method called GetObjectType in class BaseObject which returns a value of type ObjectTypeT. A BaseObject may have a list of properties which are primitive types. The class Lib represents a library of design objects which consists of Cells. A Cell may have several Views, each of specific user defined type (Netlist). An Inst represents the instantiation of a View in another View. It has two important attributes. A pointer to its Describer i.e. the View it represents, and Owner i.e. the View that owns it. In a similar manner Ports may be instantiated.
6.9.2 The CFI External Representation

The graphical representation:

Figure 6.7 shows our interpretation of the CFI Express-G schema mapped to OIS to create the external model. The central entities are denoted in bold. The other entities comprise additional relations. For simplicity, some object types have been compressed, e.g. NetScalar is represented by Net, since there is no difference in data structure or functionality.

The mapping of the port owner and the portInst owner is not according to the Express-G schema. This has no direct consequences since the owner of any port or portInst always is the view be it direct or indirect (via a bundle).
The textual representation (CDL):

Figure 6.8 shows a textual definition of CFI.

class Lib cluster private
  Attribute Name persistent pointer

class Cell general son of Lib relation many to one
  attribute Name persistent pointer
  attribute Owner persistent pointer to Lib relation one to one

class View decluster general son of Cell relation many to one
  attribute Name persistent pointer
  attribute Owner persistent pointer to Cell relation one to one
  attribute ViewType persistent
  attribute Nets persistent pointer to Net relation zero to many
  attribute Ports persistent pointer to Port relation zero to many
class ViewInst general son of View relation many to one
attribute Name persistent pointer
attribute Owner persistent pointer to View relation one to one
attribute Describer persistent pointer to View relation one to one
attribute PortInsts persistent pointer to PortInst relation zero to many

class Net general son of View relation many to one
attribute Name persistent pointer
attribute Owner persistent pointer to View relation one to one
attribute Ports persistent pointer to Port relation one to many
attribute PortInsts persistent pointer to PortInst relation one to many
class NetBundle general spec of Net
attribute Nets persistent pointer to Net relation one to many
ois_composite NBNCol = NetBundle, Net

class Port general son of View relation many to one
attribute Name persistent pointer
attribute Owner persistent pointer to View relation one to one
attribute Net persistent pointer Net relation one to one
attribute Direction persistent
class PortBundle general spec of Port
attribute Ports persistent pointer to Port relation one to many
ois_composite PBPCol = PortBundle, Port

class PortInst general son of ViewInst relation many to one
attribute Owner persistent pointer to ViewInst relation one to one
attribute Net persistent pointer to Net relation one to one
attribute Port persistent pointer to Port relation one to one
class PIBundle general spec of Port
attribute PortInsts persistent pointer to PortInst relation one to many
ois_relation PIP_C = Port, Net
ois_relation PIN_C = PortInst, Net

Figure 6.8. Textual definition of external and internal CF/ model.

The textual representation uses the CDL syntax as described in [2] and is extended with a few OIS statements. The OIS_COMPOSITE keyword defines a composite class, i.e. a class that contains a list of objects, and the OIS_RELATION defines a relation between two or more classes. The central entities and their relations are defined using the CLASS keyword and attributes of these classes are defined by the ATTRIBUTE keyword. The SPEC OF keyword denotes the specialization or inheritance of the specified class.

6.9.3 Mapping the External Schema to the Internal

The mapping from the external to the internal model occurs as follows. First the central entities in the OIS schema are mapped to a OIS+ schema as depicted in figure 6.9. To this OIS+ schema the implementation entities are added. These implementation entities consist of ancillary data and attributes. figure 6.10 shows these entities as defined for the central entity View.
Section 10. Conclusion

The tool specific internal schema
The tool programmer may adjust the internal schema with the CDL. These adjustments are tool specific and must be an addition to the general internal schema. These additions may include alternative views, new tool specific attributes, external data sources, etc.

6.10 Conclusion.

The system is based on several important design concepts:
There is a clear separation between the fine-grained and large-grained management. A virtual object manger that enables incremental data configuration forms the platform for the fine-grained design management. Providing one information model for the tool-programmer and one for the environment builder enables efficient integration of tools in a framework. The fact that the fine-grained design management is largely derived from the design objects provides a reasonable solution in the area of ULSI.

The manager described here has not been implemented. It is an investigation into the potential extension to a VOIS. A possible enhancement is an end-user browser for the fine-grained data, and enrichment of the constraints, e.g. providing triggering of events.
Figure 6.10. OIS+ representation (detailed level) of CFI class View.
Chapter 7

Tool-Tool Interfacing

7.1 Chapter Overview

The design of VLSI circuits requires large databases. VLSI design is usually done in a multiuser-multitasking environment that comprises a work station/server architecture where CAD tools for the design work are available on each work station. An interface system connects tools to a framework by providing a set of operations that the tool programmer uses when implementing the tool. The characteristics of the interface system embrace the tool programmer’s environment, run-time performance, portability of the tool, the degree of concurrency etc. The latter has direct consequences for the ease and speed at which one or more designers may concurrently work. The degree of concurrency relates to the granularity of the items locked by the interface operations since the items can only be accessed by one tool. CAD interfaces which are currently in use vary in their granularity which is generally large. The large granularity simplifies maintaining consistency of the design data but reduces the degree of concurrency.

We propose a mechanism for CAD design that enables fine grained concurrency among tools. The granularity may be defined by the tool programmer. Such a mechanism has to maintain the consistency of the data. When a tool modifies design data, no other tool may read or modify the same design data and when a tool reads a design data, other tools can only read the data. Obviously no deadlocks should occur.

We propose an arbitration protocol for tools working concurrently on shared design data. The arbitration protocol maintains the correctness of the design data by providing read and write locks at different levels of the design data hierarchy. Our concurrency model relies on certain characteristics of the interface system, namely, the capacity to perform virtual object management and to perform (run-time) partitioning of the design data. The granularity of the locked data objects is determined by the design data partitioning.

Concurrent sharing enables tool designers to think differently about tool interaction. The full potential of such a concurrent mechanism can only be evaluated when a set of tools tailored to this type of concurrency exists.

The requirements which we put on the (object oriented) binding language are slightly beyond what is currently available. Embedded mechanisms such as (pre and post) event triggering are important.
The data sharing possibilities that result from this concurrency model change the manner of inter-tool communication. A study on the relationship between the design flow, the tool synchronization and the fine grain interaction is necessary. The author wishes to emphasize that the solution presented is a small part of the total solution to the concurrency problem, namely solving the readers-writers problem in an object oriented environment, utilizing hierarchical clustering. This in itself is a condition for consistency, but is not sufficient.

The chapter is structured as follows: Section 7.2 discusses the concurrency requirements for CAD tools. Section 7.3 proposes a generic concurrency mechanism. In section 7.4 an example is given of how the mechanism works when an object oriented connectivity model of VLSI data is used and section 7.5 provides a case study. (This chapter has been compiled from [58]).

7.2 Design Requirements and Problems

7.2.1 State of the Art Concurrency
The granularity of concurrency among (VLSI) design systems varies. Most systems only support concurrency at a large granularity level. For example, if a library consists of a (hierarchical) list of cells, then once a cell has been checked-out for write no other tool may get access to the data of that cell. One of the most advanced concurrent systems is SPI, described in [59]. This system provides a form of object virtuality using remote procedure calls for inter tool communication. A so-called cell broker records the location of all cells present in the object heap of each running tool. If a tool A needs a cell that happens to be in the heap of tool B the broker checks whether the operation may be concurrent and if so transparently diverts the operation to operate on heap B. For example, a cell and the instance of that cell can be in different editors at the same time, and can be simulated as a whole. Special functions enable editing and simulating at the same time, however it is not possible to change the cell while the editor and simulator are communicating and the simulator is processing the cell. These restrictions are coupled with the fact that the cell is the entity of atomic operation. Even the advanced systems use a producer-consumer form of concurrency [60]. Some tools are considered producers and others consumers. The consumer tool waits until the producer tool finishes using the design data. This implies that feedback from the consumer to the producer is limited, usually in the form of back-annotation [17].

7.2.2 Enhanced Concurrency
In an enhanced environment, a designer would like to have a readers-writers form of concurrency in which any tool can read and write design data. Such an environment has to maintain the consistency of the data, so when a tool modifies design data, no other tool may read or modify the same design data and when a tool reads a design data, other tools can only read the data. Also no deadlocks should occur. We take an object-oriented approach to provide such an environment.
Definition
A method is activated by sending a message to an object containing the method.

Figure 7.1 presents an example from VLSI design where three tools, an editor, simulator and checker are working concurrently on the same data. Each tool reads as well as writes. The data is represented as objects in a heap. Such an environment has to provide:

1. **Heap management**: The design data must appear to be in one big heap. This can be accomplished by employing virtual data management concepts such as in [59, 17, 19, 2]. The classical form of interaction with such a heap is via remote procedure calls, in which case the heap management is performed by a special process (a server) and tools are regarded as clients. The communication overhead puts constraints on the data transfer between a tool and the heap server. Another possibility is using shared memory which enables direct access to the objects. Current operating systems set limitations on the size of the shared memory, therefore a mechanism is required for transparently loading (shared) data into shared memory. Another problem is that access to shared memory is possible only if the tools are running on the same host machine. Data sharing among different host machines is done by using at least one server.

![Image](image_url)

*Figure 7.1. Shared data scenario.*

2. **Concurrency management**: A concurrency mechanism is necessary to solve the readers-writers problem.

Definitions
A tool is locked out if its activated method is eternally waiting. A concurrent system is in deadlock when all its tools are locked out.
A child method is a method that will be executed within the body of another method (the parent).

The range of a method is the set of all objects that will be manipulated (receive messages) due to the execution of the method and all its child methods (recursively).

The range of a method can be locked in read or write mode or is free. A read lock implies that no data within the method's range is changed. A write lock implies that some data within the method's range could be changed.

3. **Data consistency**: The consistency of the design data must be maintained, thus certain operations must perform atomically on an entire group of objects.

   The system is consistent if none of the following situations occurs:
   
   - More than one method is executing within the same range that is in write lock.
   - A write method is executing within the same range that is in read lock.

4. **Tool synchronization**: The tools need to be synchronized at a conceptual level of activity. This is usually coupled with a design flow mechanism, that determines when a tool can begin execution.

The existence of a concurrent mechanism that enables tuning of the granularity level will give the tool programmer great flexibility in determining the degree of concurrency among tools. The tool programmer may, according to the tool, the method, and the design environment, be able to lock small or big design data items.

**Definition**

A parent cluster is a collection of objects or (child) clusters or both.

Our concurrency mechanism is suitable for an object heap that is partitioned into clusters. The partitions may be hierarchical to enable a good mapping to the structure of the CAD design data. This approach enables a midway between locking at a large or small granularity level. The smallest level implies locking on objects which is equivalent to the locking provided by object oriented concurrent programming languages such as [61,62]. Our view is that this level is too detailed for CAD, and doesn’t take advantage of the hierarchical structure of design data. Thus, the mechanism provides locking on a cluster level.

Figure 7.2 provides an example of how the design data is configured. Objects are clustered and relate to each other via attributes that are pointers. There are 5 clusters, c1, c2, c3, c4 and c5. c1 is the Universe cluster and contains all objects and clusters. In the example the root object of c1 sends a message to the root object of c2 by activating method m1. The method m1 sends the messages m2 and m3 and is therefore the parent of the methods m2 and m3. The range of m1 is c2, c3 and c4, therefore these clusters should all be locked as long as m1 is executing.
Figure 7.2. Example of the concurrent environment.

Definition
A concurrent system is correct if it never reaches a lock-out state and behaves consistently.

We focus on the mechanism necessary for concurrent manipulation of the design data. We propose a mechanism that ensures the correctness of the design data, found in a partitioned object heap.

7.3 Proposed Concurrency Mechanism
7.3.1 Outline of the Mechanism
We redefine the definition of range to be:

Definition
The range of a method is the set of clusters and their descendents wherein one or more objects will be manipulated (receive messages) due to the execution of the method.

The ranges of two methods overlap if at least one cluster is found in both ranges.

Any two overlapping methods can execute concurrently if they are locked for read. If the range of a method is locked for write, any overlapping method cannot execute. If a method cannot execute then it is put in the queue associated with the parent cluster of the method’s object and waits for rescheduling. A counter is kept for the number of executing read methods. This situation is similar to the readers-writers problem [60] with the following differences:

1. The locking occurs on cluster level and not individual objects.
2. The clusters may be hierarchical.
3. An active method may cause the locking of several clusters.

We introduce the concept of range locking to account for the influence of the children of a parent method outside the cluster associated with the parent method.

7.3.2 Architecture of the Concurrency Mechanism

Definitions

An arbiter is a mechanism associated with a cluster that in possible cooperation with other arbiters ensures correctness of the system.

A method is private if its range is only within the parent cluster of the object that owns this method.

A method is general if its range is not private.

An arbiter can be in state locked or state free. It can be locked if one of the following situations occurs:

1. A private method is executing in the arbiter’s cluster.

2. The arbiter belongs to a cluster that overlaps the range of an executing general method.

If a method is private, then the correctness can be managed by the corresponding arbiter. If the method is general, more arbiters participate in maintaining correctness. Thus, we can split the problem into these two cases.

Case 1: Private Arbitration

Figure 7.3 shows the arbitrator mechanism when a private message is activated. When a message is sent to an object, the activated method m first checks in at its parent cluster’s arbiter for scheduling. If the arbiter is locked, m is registered in a FIFO queue (read and write methods go into their respective queues). If the arbiter status becomes free, and m is selected, the arbiter is locked in the respective mode and m can execute. Before exiting, m sets the arbiter status to free. Since read methods can work concurrently the read queue has priority in order to reduce the waiting time. Arbiter operations are atomic, thus the arbiter will only handle the request of one process (tool) at a time.
Section 3. Proposed Concurrency Mechanism

Definitions

A method can dynamically be invisible or visible. If invisible, the method is executed without any regard to the locking mechanism.

A method is atomic if all its child methods (recursively) are invisible.

The concurrency mechanism ensures that once a method has become atomic all its children become invisible. The status of a method is determined dynamically. In plate A of figure 7.4 method m1 is the first to reach the arbiter and becomes atomic. All its children are invisible (including m2). In plate B, method m2 reaches the arbiter (since it has no atomic parent) and itself becomes atomic.
The correctness of the private arbitration (when only private methods are allowed) follows from:

- If an atomic private method \( m \) is executing in the arbiter’s cluster, all its children methods will execute in the proper order before the lock status is set to free.
- If \( m \) is a read method, other read atomic methods, private to the same cluster, will execute, as well as their children. Their order is maintained.
- Write methods with an overlapping range won’t be able to execute and will wait on the queue. Their order of arrival is maintained.
- The arbiter status is set to free by the last read method to finish execution. This within a finite period of time, since the number of read methods is finite and they are able to execute in the arbiter’s cluster.
- An atomic private method \( m \) will reach execution within a finite period of time since the queue is FIFO and the number of waiting methods ahead of \( m \) is finite.

The private arbitration problem is equivalent to the readers-writers problem.

**Case 2: General Arbitration**

General methods require access to more than one cluster. If two general methods \( g1 \) and \( g2 \) check in at all arbiters in their range in order to execute when all of them are in the front of their queues, the situation of figure 7.5 may occur. This situation causes a lockout of \( g1 \) and \( g2 \). A mechanism is needed to maintain the order among \( g1 \) and \( g2 \) in all clusters that fall in both their ranges.
Figure 7.5. Lockout situation of two global methods.

Figure 7.6. General arbiter mechanism.

Figure 7.6 shows the general arbitration mechanism. A general method $g$ enters only its atomic arbiter's queue. When $g$ arrives at the front of the queue, it is moved to the FIFO queue of the master arbiter. When $g$ arrives at the front of the master arbiter's queue, $g$ is put in all buffers of the arbiter clusters in its range (while they go on working). Method $g$ remains at the front of the master arbiter's queue until all $g$'s sent to the cluster arbiters can execute (when all their respective clusters become free). Then $g$ begins execution and the master arbiter advances its read queue or, when empty, its write queue and $g$ then leaves the front of the queue. The buffer of a cluster arbiter has priority over its read and write queues. This means that if the buffer contains a method $g$, no other method can execute in the cluster even if the cluster becomes
free.
The mechanism ensures the correctness of the system. We give a semi-formal proof.

**Property 1:** If method \( g \) is at the front of the master arbiter’s queue, then method \( g \) will execute within a finite period of time.

**Proof:** If \( g \) is at the front of the master arbiter’s queue, no other method can be at any buffer. This is assured because a method is at a buffer only if it is at the front of the master arbiter’s queue. Since the methods executing at the clusters in the range of \( g \) finish within a finite period of time and no other general method blocks a buffer, \( g \) will be able to execute within a finite period of time.

**Property 2:** A general method reaches the front of the master arbiter’s queue within a finite period of time.

**Proof:** From property 1 and the fact that the number of methods is finite it follows that a method that enters the master arbiter’s queue will reach the front within a finite period of time. From the correctness of the private arbitration mechanism it follows that a method reaches the front of the cluster arbiter’s queue within a finite period of time. If a general method reaches the front of the cluster arbiter’s queue then it will be moved to the master arbiter’s queue.
Figure 7.7. Application of the general arbiter mechanism.

Property 3: The system is free of lockouts.
Proof: Follows from property 1 and 2.

Property 4: The system is consistent.
Proof: Follows from the consistency of private arbitration and the fact that no general method executes before all its range is free and once a general method executes, it locks all its range and no other writing method can execute within this range.

The general arbitration mechanism will still be correct if more than one master arbiter is present, provided:

1. No two master arbiters hold general methods with ranges that overlap.
2. All general methods with overlapping ranges move to the same master arbiter.

Since any process (tool) executes sequentially, a process can only have one activated method, which is either waiting or executing, therefore the number of activated methods in the system is never more than the number of processes (tools).
From this follows that the maximum number of master arbiters required is equal to the number
of processes (tools). These master arbiters work in parallel since no general methods in their queues overlap. In figure 7.7 you can see that the methods \( g7 \) and \( g1 \) have overlapping range since they both affect cluster \( c2 \) and \( c3 \). In this case \( g7 \) arrived first in the queue of MA1. In MA2 there is only one active method. Since \( g4 \) does not share the range of \( g7 \) or \( g1 \) it will be executed in parallel.

### 7.4 Extending the OIS

The information regarding the method categorization is added to the OIS schema as presented in figure 7.8. A specialization of \textit{method} is \textit{shared-method}, which is related to the (cluster) hierarchy.

![Figure 7.8. OIS schema including method categorization.](image)

The CDL is extended to specify the type (Read/Write) and the Range. The \textit{shared-method} entity is a specialization of \textit{method}. In our case, it only applies to virtual objects, but being related directly to \textit{UserClass} is the general case. The relation to \textit{Hierarchy} provides the information regarding the range of the method.

### 7.5 Concurrency Applied to VLSI Data, a Case Study

Various systems now use an object oriented approach and manage an object heap [17, 19, 2, 44] as the temporal store of an interface system. Usually a procedural interface (e.g. C binding) is used and in some cases an object-oriented language (e.g. C++ binding) may be used. We base our discussion on the latter. In this chapter we shall apply the concurrency mechanism to an interface for VLSI design data that uses a Netlist data model (see Figure 6.6),
based on [17]. It is a good example for our reasoning since it clearly shows the hierarchical levels of design data which suggests that partitioning may be done below the cell level. Figure 7.9 provides a legend for the types used in figure 7.10. It is slightly more detailed to accommodate for usage in the following chapters.

![Diagram of CFI Type Legend]

**Figure 7.9. CFI Type Legend.**

We show a simplified view of the objects as they might appear in the object heap in figure 7.10. The stipple lines denote partitions.
A Lib instance and its Cells are clustered since the number of instances is usually not large. The proposed clustering mechanism allows all tools to access the Lib instance and the Cells. In a traditional mechanism the Cells would have to be separated and a problem would still exist for manipulating the Lib instance. The View instance is in a separate cluster since it is wise to enable fetching the Netlist view from the data-base without fetching the Layout View as well, and if both are in the heap they can be manipulated concurrently. In most systems the partitioning ends with View, however it is advantageous to cluster (view) Instances and Nets as well. This enables only minimal data of a View to be fetched from persistent store, i.e. the data necessary for an instantiation. Moreover, different tools may (in certain cases) concurrently work on e.g. Nets and Ports without waiting.

A more complex example is the following method:
NetBundle* View::CreateNetBundle(StringT name,
StringT members)
{
  opiEntry();
  .
  For each member
    NetScalar = CreateNetScalar(netbundle->
    ObjectId, netname, error);
  .
  opiExit();
}

Figure 7.11. Skeleton C++ definition of class View.

Here the method CreateNetBundle is a parent of CreateNetScalar. Note the functions opiEntry() and opiExit. These enable the atomic locking and freeing of arbitrators and are inserted in the code by a pre-processor during compilation. CreateNetScalar might also be activated as a top level method in which case it would become an atomic method, and the range appropriate to it would be locked. However, when activated as a child method the atomic lock is set on the parent (CreateNetBundle in this case) and a different range is locked, which contains the range of CreateNetScalar. The child method will be transparent to the locking arbitrators and their composite methods will be executed with a small overhead.

The problem of determining the range of a function remains. At the moment the range is determined manually by inspection of the method, the object type(s) upon which it operates and the partitioning of the heap. By carefully looking at the problematic pointers, i.e. those that cross cluster boundaries and by beginning from leaf methods and progressing towards the most composite methods the range may be determined. The clusters are typed according to the type of the root object of the cluster. For example the range of CreateNetBundle is cl. 4 and cl. 2, since the View object must be set for ownership. The range determination could be automated, for example by traversing the object pointers of a type. In our implementation the ranges of the methods are defined (before compilation of the tool code), in the Configuration Data Language (CDL) which is used for defining the partitioning of the data.
Chapter 8

A Test Environment

8.1 Chapter Overview

An important part of a programming environment is a system for testing the behavior of the software under development, known as a test harness. The ECAD (VLSI) environment is complex, since it includes numerous tools coupled to a framework. We wish to address the following questions:

What is the most effective way to realize the test harness?

We concluded that the same framework that supports the ECAD tools should be used to support the test harness. This is the only system powerful enough to support the complex management required for an effective test environment.

How should a test be modeled?

Since the framework is an object oriented DBMS, the natural choice was to model tests as objects in a hierarchy.

This chapter is structured as follows: Section 8.2 summarizes general problems involved in testing. Section 8.3 gives the design philosophy and testing concepts. Section 8.4 gives the design considerations and section 8.5 the design of the test harness. Section 8.6 presents the implementation of the harness and we conclude in section 8.7. (This chapter has been compiled from [50, 63, 64]).

8.2 Introduction to the Problem

An estimation published in [65] puts the price tag of testing at 50% of the cost of system development, thus automating the test process is an important way to slash costs. Test automation may be split into three categories [66]:

1) Automating the administrative side of the testing activity, e.g. looking after printing and the identification of tests, storage of sessions etc.

2) Automating the mechanical side of the activity, e.g. the running of the software under test within a test harness, the application of test data to it and the comparison of actual results with
those expected.

3) Automating the generation of test cases.

This chapter describes a test environment that covers the first two categories. The last category is considered the most difficult to automate. However, before attempting automatic test case generation, it is essential to have a good test harness.

The investment of resources in test automation appears worthwhile because the development of frameworks and tools for VLSI CAD is an incremental process. New releases deviate slightly from their predecessors. This is enforced upon the vendors by clients, due to need for upward compatibility. However, the complex interaction between sub-systems often compels one to test the system as a whole even if altered only slightly. Moreover, the same validation must be performed on several machine types.

Currently, state-of-the-art ECAD systems known to the author, do not have a test harness. Testing of frameworks evaluated within the JESSI Frame evaluation sub-project of ESPRIT has been carried out manually [67]. Automating the testing has several advantages:

1) Quality: Circuits of relatively larger order (which consume more time) can be used as inputs for tests since testing becomes less labour intensive. This is likely to increase the quality of the test, since some problems in ECAD software do not occur when small circuits are used as test data.

2) Reliability: manual testing is error prone due to the longevity and complexity of certain operations.

3) Robustness: The amount of tests and sorts of test can be increased which is likely to make the system under test more robust.

4) Replication: the test can be repeated accurately, which enables a complete reconstruction of the whole design environment, up to the point where an error occurred.

5) Measuring: important run-time data can be gathered and stored, for analysis of performance, such as memory and CPU usage.

8.3 Design Philosophy and Testing Concepts

8.3.1 Design Philosophy

The design of our test harness is based on two main concepts:

1) A test harness should be an ancillary component of the CAD framework.

2) The test data should be modeled as objects.
These concepts have the following advantages:

1) By building on top of the framework itself, a powerful test environment may be produced with minimal implementation effort.

2) The resources used to operate the system (Disk and RAM) are minimal.

3) The (framework) client may use the harness for local testing.

4) The framework may be shipped with a test package which enables on-site testing.

5) Modeling tests as CAD objects on the same framework improves the learning curve of the harness for the client.

6) A formal channel exists for clients to communicate problems to the vendor, since a complete test suite may be defined as CAD objects which may be run at another site.

8.3.2 Testing Concepts

Definitions

Verification involves evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous stage.

Validation involves testing software or its specification at the end of the development effort to ensure that it meets its requirements.

Producing reliable systems requires a well-planned, comprehensive application of several techniques by many players throughout the development life-cycle, collectively known as Verification & Validation (V&V) [68]. The Test phase is applied to both verification and validation. Testing involves several tasks [69]:

1) Build the test harness.

2) Generate the inputs.

3) Determine the expected output, known as the test oracle problem.

4) Execute the module under test on the selected inputs, monitoring system/module behavior.

5) Compare the actual outputs to the expected outputs.

Definitions

An entity in testing is the test case, which is a collection of input data, output data, comparative data, and a sequence of functions to be executed.

A combination of test cases to be executed in sequence forms a test suite.

System testing is an aggregation of tests suites that together exercise the system as whole.
There are two types of system testing: white-box, in which the test data is related to the structure of the program, and black-box (or functional), in which the test data is constructed from the specifications.

White-box testing is effective for probing for weaknesses in sensitive regions and is suitable only for the system vendor, while black-box testing requires no knowledge of the internal structure and therefore is also suitable for the client. Within the validation category we focus on black-box system testing.

8.4 Design Considerations

8.4.1 The NELSIS VLSI Design System

The NELSIS VLSI CAD system consists of the framework, (introduced in chapter 3) and a set of IC design tools integrated using the DML. These tools support the development of IC’s.

The tools can be classified into three packages:

1) The Simulator package, including a digital mixed level simulator (SLS), an analog simulator (SPICE), graphical user interfaces and interfaces with standard network descriptions (i.e. EDIF).

2) The Layout package, including an interactive layout tool (DALI), tools for layout generation and interfaces with standard layout descriptions (i.e. GDS2).

3) The Verification package, including layout rule verification, an extractor (SPACE), and a layout-to-circuit verification tool (MATCH).

Apart from these kernel packages a number of other specialized IC design tools are available.

8.4.2 General Considerations

In section 8.3.2 a list of testing tasks was presented, the first being: "build the test harness"; but how? We concluded that the architecture of the harness depends on the type and quantity of test data to be managed. We studied the list of additional tasks and from [69] it was clear that the most critical testing task is item 3, i.e. defining the expected output. Apart from being the most time consuming task, the reliability of the testing is directly correlated to this task. Our solution was based on the resources at our disposal, namely several fairly large VLSI designs, completed and simulated for correctness. Some of these have been fabricated and tested. Since these designs were built with the NELSIS CADSystem, we had access to all ancillary data, such as simulation files and complete design databases. This data represents many years of work and is sufficiently reliable.

Definition

(In the context of testing) an environment is one or more databases (NELSIS projects) and possibly related ancillary data in the form of files (in the UNIX file system).
The domain of the output was chosen to be an *environment*.

Having defined the expected output, the input definition (task 2) was decided to be in the domain of the output, thus a basis for a hierarchy of tests is formed which is essential for coping with modular construction of test suites. Executing the module under test (task 4) is therefore performing part of a design trajectory.

**Definitions**
A pre-environment of a test is the environment that exists before executing the test. A post-environment of a test is the environment created through execution of the test.

Thus, a test case consists of a pre-environment, a design trajectory and a post-environment, and a state is the set of environments that exist in steady-state (between transitions). In figure 8.1 the pre-environment $E_{pre}$ of a certain design trajectory $T$ (the test) is transformed (by executing $T$) into a new environment, the post environment $E_{post}$.

\[
\begin{array}{ccc}
E_{pre} & \rightarrow & T \rightarrow E_{post} \\
\square & \quad & \circ \\
\end{array}
\]

![Figure 8.1. Basic state transition between environments.](image)

The state transition can become more complicated:

1. **A pre-environment may be the input to several test cases.** Since a test run causes a transition, the harness must be able to save the pre-environment for re-use with a different test-case. This is explained in figure 8.2. Plate A depicts an environment ($E_{pre}$) used as an input for two tests (T1 and T2). In the time domain (Plate B) this means that first a copy ($E_{pre}'$) is made. Execution of the tests is only possible after completing this operation.
2. A test case may use a number of post-environments as its pre-environment. In this case, the test may only start when all generating tests have completed, (since the different post environments together form the pre environment for the test that is to be started). This is depicted in figure 8.3. In Plate A, a test (T3) needs input from several output environments. This can be seen as one pre-environment. Therefore in the time domain (Plate B) execution of test T3 may only be performed after successful completion of tests T1 and T2.

### 8.4.3 Specific Considerations

Given the proposed state transition mechanism, a number of specific considerations are listed:

1) It should be possible to reconstruct an environment. This is justified by the fact that a test person should be able to manually examine any environment created during the execution of a test suite, after the test has been driven, in order to obtain additional information (if necessary). Test cases are extremely time consuming (e.g. simulation) and therefore it is not acceptable to start a test suite from scratch.
Section 4. Design Considerations

2) It should be possible to *duplicate* an environment. We studied the problem and concluded that NELSiS design projects may be duplicated and relocated without loss of functionality or data. If this were not possible, our test method would not be practical.

3) It should be possible to *initialize* an environment. Often an initialization needs to be performed on a pre-environment to make it more suitable for a particular test run.

4) Tools for *inspecting* and re-setting environments and results, defining test cases etc, should be present.

**Example**

We demonstrate how the environment mechanism is related to an automatic test-suite execution in figure 8.4.

![Diagram](image)

**Figure 8.4. A test suite mapped to states and transitions.**

The rectangles and circles denote states and transitions (test runs) respectively. At the bottom left we have environment E1 which is the initial state. When test run T1 has passed, environment E1 is transformed into E2.

Because E2 is the pre-environment for two tests, T2 and T3, the environment must be sustained (rectangle denoted by S). When the tests T2 and T3 have passed, two new environments are created, E3 and E4. At this point the sustained environment may be erased. The two post-environments of T2 and T3 together form the pre-environment for test T4. When a test fails the (incomplete) post-environment is sustained by default. This is denoted in the figure by the dashed lines, leading to the state called non-complete.
8.5 Design of the Test Harness

8.5.1 General Overview

The Automatic NELSiS Tester (ANT) consists of a set of test tools built on top of the NELSiS framework which enables the design and management of test cases, and their automatic execution. Test cases are mapped to design objects (called test-objects for compliancy). Thus, in analogy, a test suite may be mapped to a hierarchical object. Using the frameworks design system interface the user can browse through the test hierarchy to examine and manipulate the status of the tests. A special tool provides the user with a graphical interface to the ANT system, which enables easy invocation of the test tools with the necessary parameters and displays essential actions on a monitor.

The test harness manipulates a pilot project which contains all information concerning the design and execution of tests. The test data consists of ( VLSI ) design data and related design projects. Test cases are defined by the user in a format that can be processed and stored in the pilot project. ANT executes a test suite by traversing the hierarchy. It gathers information on the way, prompts the user for additional information if necessary and reports failed tests. It offers various tools to rectify tests, reset test sequences, and restart execution.

8.5.2 Utilizing Hierarchy for Test Sequencing

The tool execution sequence occurs in the opposite direction of the hierarchical representation. The lower level test objects are the child tests of the higher test objects (the parents). When a parent has two or more children, it implies that the child tests must have passed their test before the parent test can be executed. If a test object has multiple parents, ANT creates a duplicate environment, which enables both parents to execute without loss of consistency. Once the test has progressed to a higher level, ANT may destroy environments of a lower level, to avoid data explosion.

Example

Figure 8.5 illustrates a simplified test suite. The top level object, T_NELSiS, represents the testing of the entire CAD system. It splits into two objects, T_Frame and T_Tools. These represent different test groups: the first being common framework aspects and the latter being design tools.
Figure 8.5. A simple test suite mapped to hierarchical CAD objects.

Under T_Frame are tests for the user interface and the browsers. Under T_Tools are the tests for the tool space. In the figure, the test T_browsers cannot be executed before the tests T_Vbrowser and T_Hbrowser have passed. Thus, if an environment created by the execution of T_space is used by T_space_f, then a copy of the environment is made for the execution of T_space_h. Once T_space_h and T_space_f have passed, their pre-environment may be destroyed, i.e. T_space’s post-environment.

8.5.3 Design and Test Modes
The user may use the harness in two working modes: design and test, as shown in figure 8.6. In plate A, each circle represents a major phase in the design cycle (as most commonly carried out). The major operations performed in the particular phase are shown on the left, and the tools used during that phase are presented are shown on the right. In plate B, each circle represents a major phase in the test cycle, the associated major operations (left) and the tools involved (right). These tools will be discussed in the next section.
The Design Cycle
A test definition is constructed of the initialization, execution and checking components. The definition is not likely to be correct after the first installation and execution, it is a cyclic and iterative process. During design, each phase may be checked separately. When developing a test-object that is hierarchically dependent on child test-objects, the pre-environment may be sustained so that the test can be reset quickly.

The Test Cycle
During the test cycle there are two test options. The first is executing each phase separately which enables checking each phase in detail, and is used during the design phase, by the test designer. However, the tester may use this option when needing more detailed data on a test. In general, during testing, the second option is used, by invoking the tool testTO. TestTO runs the separate test phases in sequence. The definition of the sequence may be modified to enable setting extra parameters. When invoked in the hierarchical mode, testTO runs a complete hierarchical test.
8.6 Implementation of the Test Harness

8.6.1 Data Schema

Figure 8.7 shows the data schema of a pilot project describing all the inherent aspects of the test-objects. It differs only slightly from the standard NELsIS data schema [10]; one leaf attribute has been added, TOEnv and Pilot has been made an attribute of Module. We shall now provide an interpretation of the schema.

![Diagram showing the ANT data schema](image)

Figure 8.7. ANT data schema.

The data schema is defined according to the OTO-D variant used by the NELsIS system [10]. Rectangles denote types. A line from the bottom edge of a rectangle to the top edge of a rectangle implies that the bottom type is an attribute of the top type. A line from the bottom corner of a rectangle to the top corner of a rectangle implies that the top-most type is a specialization of the bottom type (inheritance).

**ViewType:** There are five ViewTypes. During the design phase, the complexity of a test object is broken down into separate sections, each represented by a ViewType.

1) **INIT** consists of DO’s that contain initialization data: input files and initialization scripts.
2) **EXEC** consists of DO’s that contain execution data: names of expected output files and execution scripts.
3) **CHECK** consists of DO’s that contain check data: reference files and check scripts.
4) **TO** consists of DO’s that contain organization data: report scripts and documentation.
5) **TEST** consists of DO’s that contain the result data that is gathered during the execution of the test phases.
Hierachy: Hierarchy only applies to the TO ViewType. The Hierarchy registers the parent-child relationships between test objects.

Equivalence: The ANT system maintains two types of equivalence, therefore Info may have two values. The first represents the normal NELSIS equivalence, i.e. one DO (DesignObject) is derived from the another DO through a tool manipulation. The latter equivalence is used to model a test object set. Each equivalence instance relates DO’s from different ViewTypes that together form an entity.

DesignObject: The DO has the attributes: Vnumber, Vstatus, Module, Imported and Date. Vnumber is the version of the DO. Date contains the test installation date. Module is a composite attribute containing the attributes: LastVersion, Name, ViewType, Project, TOEnv and Designer. LastVersion contains the last installed version of the test. Name is the name of the test. TOEnv is only meaningful for DO’s of the TO ViewType. It may have the values: Saved, NotSaved. Saved implies that the post-environment of the test object has been saved.

Import, Export and Library are handles to test objects in other pilot projects. The Import mechanism is not meaningful for the TEST ViewType. It is used to (hierarchically) import a complete test object.

Project: Type Project is used in two contexts, the normal attribute of Export, and the additional attribute of Module. The latter associates a design project with the test object.

Vnumber & Vstatus: In the TO ViewType the Vnumber (version number) enables defining many test objects of the same name. The meaningful Vstatus (version status) values are Actual, Working & Backup. When the ANT system is manipulated in the Design mode, the Working version is selected by default. When in the test mode, the Actual version is selected.

In the Design category the versioning mechanism is the same as in the Framework. In the Test category, all objects are of the Derived status, since they are the result of extraction based on the Design ViewTypes.

Transaction: Transaction registers the tool interactions with the database.
8.6.2 The ANT Tools

Figure 8.8 shows how the tools are incorporated in the framework.

"Dog-eared" rectangles represent files. Rectangles represent directories. Circles represent ANT tools. In the center, the five rectangles represent the view-types of the pilot project. The arrows show the direction of data flow during tool execution. Arrows connected to the large rectangle that engulfs all the view-types imply that the data may flow to or from several view-types. A stipple lined arrow implies that a tool manipulates the data in the view-type pointed to, but does not necessarily transport it (e.g. rmTO).

ANT provides three categories of tools: General purpose, Design and Test.

The general purpose tools are antui, mkPilot, rmPilot, tsui, impTO and rmTO. The first five tools are not shown in the figure. They include test pilot project creation and deletion tools and user interfaces. With impTO test objects from other test pilot projects can be included (called). rmPilot installs test objects in the database.

The test tools, envTO, execTO, checkTO, reportTO and testTO, are used for (automatic) test suite execution.

These tools control the environment of tests (Save, (Re)Set), perform test runs (in a hierarchical order), check the results and generate the test report.

The design tools are extra tools needed for the definition of the test cases and include the tools defTO, defSHS (define Shell Script), defIAS (define Inter Active Script), dbEdit and rmTO.

They extract test object definitions, control (interactive) test scripts from the database for editing or long term storage, or (in case of new data) provide template definitions. rmTO
removes test objects from the system.

A typical test runs as follows:
During the install phase the tool mkTO parses the (possibly hierarchical) test object definition script and stores the test cases in the streams organized in the different view-types of the pilot project. envTO invoked in mode SET, creates the initial state for each test in the workspace, a directory in the pilot, in which the actual test run takes place. execTO, guided by the information in the EXEC view-type, performs the actual automatic or interactive test run. Results are stored in the TEST view-type. checkTO, guided by the information in the CHECK view-type, compares referential data from the CHECK view-type against the test results. Then reportTO may generate a report.

After each test, the data in the workspace, or post-environment image, may be saved in the TEST view-type using the tool envTO in SAVE mode. The image can be recaptured (if so desired) using envTO in RESTORE mode. When envTO is invoked with the RESET mode, the pre-environment of the test is restored by copying the post-environments of all its sons from the TEST view-type to the workspace. The database (file system) architecture of a test pilot is depicted in figure 8.9. Note that complete databases may be stored within a NELSIS design-object. This proves the importance of supporting the large grain interface as depicted in figure 1.2, even if a fine-grained interface exists.

**Figure 8.9. Mapping of ANT to the NELSIS storage system.**
8.7 Conclusion

Results

ANT is presently used at the Delft University of Technology where it is being tested and improved. It is used for regression testing of the NELSIS Framework and tool releases and for reporting problems back to the framework development group. The harness is further used for OPI module testing, and as a basis for NTE [70], an integrated training system for NELSIS users, (under development), built on top of the NELSIS Framework, in which test cases are replaced by training sessions.

A test suite has been implemented for the NELSIS CAD System [71, 72]. It consists of some 133 test cases and requires around 3.5 hours to execute (batch mode) on an HP300.

Suggestions

In our opinion framework consumers should demand a test harness together with a test suite to be shipped with the framework (system). Customers studying the extent of the suite and executing it, may gain valuable insight into the reliability (and functionality) of the system without performing extensive in-house evaluation. Moreover, much better feedback of problems is possible.

Such a harness also enables formal communication between the development and the validation group. This is extremely important, since a key element to the effectiveness of testing is that a program be observable and controllable [73], which result in a program being easily testable. These characteristics can be introduced by the development team only.

Future work

The automation of interactive mouse driven graphical tools is a major problem. At the present, ANT lacks such a mechanism and pops up communication windows that guide the user (in this case manual test execution). This problem has been solved in [74] by utilizing the X record and playback monitor XTM. Being a public domain tool we believe this to be a feasible solution to our problem.

The second problem is the development of test cases. Although ANT provides support for this procedure, beside being a time consuming process it is primarily a theoretical problem. Only raw estimations have been made about the test coverage of the test cases, which influences the soundness of the system test [66]. We are active in this research area, and are defining a methodology for automating the structure of a test-suite [64].
Chapter 9

A Conventional CFI Interface - Case Study

9.1 Chapter Overview

CFI is an international organization for standardizing tool and DBMS interfacing. Its tool - DBMS interface, known as DRPI, is a document defining in great detail a set of C functions for performing ECAD based on a NetList connectivity schema. This project has created a unique opportunity to make a comparison between a conventional interface and a VOIS, by implementing the specifications in both interfaces. In our view, many system developers still underestimate the problems involved in DBMS mapping and the time and effort required to bridge the gap. The study of the design of a conventional CFI interface brings out the problems related to persistency that the tool programmer faces when implementing the DRPI functions. The study of the VOIS implementation of the CFI demonstrates the power of such a system to deal with the persistency problems. In this chapter we first present the CFI, and then go through the steps of designing the conventional interface as it actually occurred. (This chapter is compiled from [75], a report summarizing the work involved in designing and implementing a CFI-DRPI compliant interface system, that is incorporated on top of the NELSIS Framework).

9.2 Problems Coupling CFI-DRPI with NELSIS DMI

An explanation of the CFI interface and the data-schema of the elementary objects has been presented in section 6.10.1. The first step is to perform a feasibility study in order to determine how large the interfacing gap is and how this is to be bridged.

9.2.1 Main Differences between CFI and NELSIS.

The data schema used by CFI (Express-G) and the NELSIS variant of OTO-D do not match. For the purpose of this discussion, the NELSIS schema of figure 8.7 may be referred to. By comparing the CFI with the NELSIS schema the following differences can be found:

CFI models all entities in terms of object types. NELSIS uses object typing to model the relations and status of a generic object.

In CFI the ViewType is a dynamically defined attribute, while in NELSIS it is a static definition and has consequences for storage.
In CFI the scope of a name varies according to object type and according to relation. A name must be unique within the owner of the object. In NELSIS the naming of design objects is scoped per ViewType.

In CFI Inst is just another object, and the hierarchical relationship is captured through the describer attribute. In NELSIS instantiation is not modeled as an object but as a parent-child relation in the Hierarchy entity.

CFI enables the selection of instances (alternatives like slow, typical, fast) within the scope of one View. In NELSIS the concept of alternatives does not yet exist.

In CFI all object types may contain design data in the form of objects which are called Properties. In NELSIS a conceptual difference exists between design data and metadata. Meta data contains relations and information about the design data.

CFI (version 0.7) does not support versioning. This means that instances can be replaced at each level of the hierarchy. NELSIS not only supports versioning but enforces it through Module, since all objects with an identical Name and ViewType are versions.

In CFI any library is accessible by name. Reachability of projects in NELSIS is implicitly via Import/Export relations.

CFI uses the concept of a volatile object heap, i.e. a library can be created and manipulated (in memory) without actually writing data to persistent store (on disc). The persistent image (part of it) is only updated when a specific save command is given. NELSIS only manages changes to persistent data and access information to that data.

The CFI volatile heap has one entry point via the CFI-Server. To avoid corruption of the heap the tools must be synchronized using an additional communication facility. NELSIS is a multi-user multi-tasking system and does not require tool synchronization for protecting consistency.

All these differences form the basis of the problems that arise in the different kinds of mapping of CFI to NELSIS that we will discuss in the next sections.

9.2.2 Mapping Possibilities
There are four basic mapping possibilities divided into two groups: Complete mapping and Partial mapping. In each of these groups one may choose between binding CFI to NELSIS, or vice versa. Therefore we may study the following four cases:

Case A: Complete mapping binding NELSIS to CFI.
Case B: Complete mapping binding CFI to NELSIS.
Case C: Partial mapping binding NELSIS to CFI.
Case D: Partial mapping bending CFI to NELSIS.

We shall use OIS to depict the data schemas.

**9.2.2.1 Case A**

From the CFI users point of view there should be no distinction between meta- and design data since CFI tools work with objects. To accomplish this a data schema would have to be defined describing the CFI data schema in the variant of OTO-D as used by the NELSIS system. This implies capturing all the relations between the instances as (partially) shown previously in figure 7.9.

**Advantages (CFI end-user)**

Very comfortable for the CFI user. The user can visually browse and query all parts of the design data in the most natural way.

**Disadvantages (CFI user)**

None

**Implementation Problems**

The NELSIS kernel is geared towards the manipulation of so called Design-Objects as defined in the standard NELSIS data-schema. Any deviation that changes entities of this schema will necessitate some rebuilding of the NELSIS kernel and rewriting (parts of) the documentation. This mapping would change all key entities of the NELSIS data schema. The amount of metadata will grow with a factor 20 - 200, depending on the average complexity of the cells.

A volatile heap is not feasible.

Due to the complexity of the summarized problems it was decided not proceed in this direction.

**9.2.2.2 Case B**

To better conform with the NELSIS data schema we could model CFI objects as one central Object type. The schema shown in figure 9.1 is enough to capture the CFI schema. In this case all relations are modeled via equivalence.

**Advantages**

The data schema conforms with CFI. The user can visually browse and query all parts of the design data.

**Disadvantages**

Not all object types are visible in the schema and queries have to be carried out piece-meal.

**Implementation Problems**

Adaptations are still necessary to the kernel, but less severe than in case A. The amount of metadata, as in case A, would grow considerably and the volatile heap requirement of CFI is not met.
Solutions
A proposed solution was to split the data into metadata above and below View object level. The data describing the view itself is considered to be metadata located above the view object level. Each View object would have its own metadata tables in working memory, that are read when the View is checked out.

9.2.2.3 Case C
The alternative for the solution in case B was to create an object heap and map all objects above view to metadata, and all objects below view as design data. The need for an object heap was the direct consequence of the CFI specifications, that allows a complete memory resident library to be created and manipulated without being saved persistently. (see also section 9.3.3). The concept of design data and metadata is upheld.
In this case a choice is made of what level do we consider metadata. View is the most natural level because it corresponds to an electronic component entity. The only object below this level that belongs to the metadata is the Inst object, since this object represents the highest level of inter-component relationships (the design hierarchy). Thus the mapping is reduced to the mapping of the CFI schema shown in figure 9.2.

Advantages
The CFI user can visually browse and query the top parts of the design data in the most natural way.

Disadvantages
The CFI-schema is broken into two parts, and only the top part can be browsed and queried.

Implementation Problems
Same as in Case A except for the amount of metadata.
Section 2. Problems Coupling CFI-DRPI with NELSiS DMI

Figure 9.2. Overview of CFI top level objects.

Figure 9.3. OIS of CFI top level objects.

Solutions
We offer no solution due to the gravity of the problem.

9.2.2.4 Case D
Use the OIS as proposed in case B (figure 9.1), but Object/Type is limited to Lib, Cell and View and EType can be Instance or Ownership.

Advantages
The data schema conforms with CFI.
Disadvantages
Not all object types are visible in the schema and some queries have to be carried out piece-meal. The user can browse and query the top parts of the data schema only.

Implementation Problems
Adaptations still necessary to the kernel, but less harsh than in case C.

Solutions
There are several solutions to problem 1. The first is adapting the NELSIS kernel to accommodate this schema. The second is to leave the NELSIS data-schema intact and move the CFI schema further towards the NELSIS data-schema. The second solution has two variants, D1 and D2, which we will discuss in the next sections.

9.2.2.5 Variant D1
In this variant we model Lib, Cell and View as a DesignObject (figure 9.4).

---

Advantages
The Name and ViewType attributes can be stored at the most obvious places. The Properties of the high-level objects (LibDO, CellDO) can be stored (as design data) in the Design Object. All the relations can be stored in the Meta Data: The relations between Lib and Cell, and Cell and View can be stored as equivalences. The kind of the relations can be stored in the Info field. The Inst can also be stored in the equivalence, but when the Hierarchy is also used (stippled in figure), it is also possible to store the Inst there. Alternatives can be modeled (See Variant D2)

Disadvantages
An extra tag has to be attached to the DO, called ObjectType (dashed in the figure), in which the
DO Type (Lib, Cell, View) is stored. The metadata increases, depending on the average number of Views in a Cell, with a factor 1.5 to 2.0. The number of object directories increases with the same factor as mentioned above. Some of the queries have to be carried out piece-meal.

Problems
Names of Libs, Cells and Views have to be unique in the scope of the whole library. In CFI one may change the description of an instance. In NELSI S this implies that it must be possible to replace one DesignObject with another in the hierarchy relationship. The standard method is using the tool instcell (install), which is closely coupled to the version mechanism. The NELSI S ViewType is not dynamically configurable.

Solutions
The name attribute of Lib contains a NULL string, because the name of the Lib can be retrieved from the Project. The name attribute of a cell contains the Cellname, and the name attribute of the View contains the string "Cellname"<delimiter>"Viewname". The static viewtype problem can be solved by adding an additional attribute CFIViewType to DO (or change the kernel so that ViewType becomes dynamic).

9.2.2.6 Variant D2
In this variant the Express-G type View is mapped to the NELSI S DesignObject, the Express-G type Cell to the NELSI S Module and the Express-G type Lib to Project, as shown in figure 9.5.

![Diagram of DO, Hierarchy, Equivalence, Name, Vnumber, Vstatus, CFIViewType, Module, Name, ViewType]

Figure 9.5. OIS variant D2.

Advantages
The amount of metadata is 20-50% less than in variant D1. There is no need to do equivalence metadata management for relations between Lib and Cell and Cell and View. There will be some minor changes, there is no need to change vital parts of the NELSI S data schema.
Disadvantages
Another issue is whether the meaning of Design-Object and Module are acceptable in the context of CFI. In principle a CFI user would only want to know about the object types of Express-G. The top level metadata is not complete from the users point of view.

Problems
The relations between Lib and Cell and Cell and View need to be captured in another way. Where do we store the top-level design data? ViewType belongs to Module (the Cell) and not to DO (the View). The scope of a name is tied to the Cell (alternative problem).

Solutions
One design object is created automatically by the system called LibObject which contains the relations Lib/Cell/View including their respective design data. This object is always checked out when the CFI function initPl is called, thus initializing the heap. An additional attribute CFI/ViewType may be added to DO (or change the kernel so that ViewType becomes dynamic).

9.2.3 The Object Heap
The Express-G DRII objects are kept on a volatile heap managed by a CFI server. All actions that effect these objects and their inter-relations are not persistent until either a View or a Library is saved. This object heap contains (captures) both meta- and design data.

The consequence of having an object heap is that in fact two data-bases exist, a volatile one and a persistent one. This may cause confusion, since these two images are not consistent. For example: when a View is fetched from the NELSIS data-base (getView), it is also checked out by NELSIS, and the action is recorded in the metadata. However, if an object is deleted (deleteObject) then this does not effect the NELSIS data-base since according to the CFI specs, all actions on the heap are only to be made persistent if a save command (saveObject) is given. Also, if a purge command is given, then the object heap is effected but not the data-base.

9.2.4 Creating Libraries
CFI libraries are created through the use of functions. NELSIS uses the tool 'mkpr' for this purpose. To integrate the ability to create libraries we only have to encapsulate the NELSIS tool. This implies that the tool should always be reachable and must have a fixed name. If the DMI were extended with a dmCreateProject function, this would be no problem.

9.2.5 Library Placement
A library should always be reachable 'by name' according to the CFI specs. This implies that a record should be kept to indicate where a given library can be found. A file containing this information should be created in a 'safe' place. This would preferably be "nelsis/CFI/cfi_admin. If the server is started by user nelsis this would be no problem and normal users would not be able to modify the file.
9.2.6 Storage Aspects
All data in the view hierarchy is stored in streams in its DO.

Advantages
The data is immediately accessible and traversable.

Disadvantages
A view may contain very large quantities of data that is rarely used (for most tools).

Future development
An incremental load and save mechanism. Due to the complexity of this problem this capability will not be implemented in the first prototype.

9.3 General Architecture

We begin with a top down study of the CFI DRPI, in order to understand what kind of software modules are involved in the design and how they should interconnect based on the characteristics of the data management.

Figure 9.6 shows the general CFI configuration.

Figure 9.6. The general configuration.
The CFI manager communicates with the DMI on the one hand and with the CFI server on the other. In principle, the manager may also be linked to a tool without the server. In the current configuration only single tool interaction is required. One tool writes design data via the server and signals it is finished. Only then is the read tool allowed to read the design data.

According to the specifications a so-called CFI-DRPI server is responsible for performing all the CFI functions.

9.3.1 The CFI Server
Figure 9.7 shows the coupling between CFI tools and the common NELSIS tool communication configuration. To the left we see a CFI tool tagged T3. T3 only "sees" the CFI server and has no direct communication with the NELSIS framework. The server controls the manager via the CFI interface. The current interface specification is that all tools communicate with one server per host machine. Note that from the point of view of the NELSIS framework the data manager is just another tool requesting services from the framework via the DMI.

![Diagram of CFI server](image)

Figure 9.7. Server diagram.

The server is linked to what we call the CFI Manager. This is the component that we must design.

9.3.2 The CFI data manager
At this stage there is no assumption made of how the CFI manager is constructed, it is simply a black box. We now take a look at the functions, i.e. input and outputs, pre-conditions and postconditions in order to obtain some knowledge of what must be going on in this black box.
9.3.2.1 Data Availability

Figure 9.8 shows an example of one of several tables in which CFI function definitions are analyzed. Based on the specifications related to the input, output and whether the data should be made persistent or not, we can define what data should be present at certain locations before and after a function call.

We define three separate data repositories: in the application itself, in the CFI data manager and in a persistent repository (managed by a framework).

<table>
<thead>
<tr>
<th>cfiSaveObject</th>
<th>Application</th>
<th>CFI Manager</th>
<th>Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectType</td>
<td>before</td>
<td>after</td>
<td>before</td>
</tr>
<tr>
<td>Lib</td>
<td>OldLib</td>
<td>—</td>
<td>OldLib</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>View</td>
<td>OldView</td>
<td>—</td>
<td>OldView</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9.8. Availability of data types for the save function.

The availability tables determine the time and repository location of the data related to the specified function.

9.3.2.2 Conclusions based on Data Availability Tables

From tables such as in figure 9.8 it follows that the CFI manager must have its own independent storage component since not all data types are available to a tool. From the save function we can derive that a database is necessary for a persistent storage component. This leads to figure 9.9.
Figure 9.9. The CFI manager - basic modules.

The CFI manager consists of 3 basic sub-components, the interface, the data manager, the data repository and has the following basic components:

An interface layer that contains the CFI functions.

The CFI data manager for managing the data under the control of the CFI module.

The CFI data repository for containing the data under the control of the CFI data manager.

All functional components are capable of interfacing with the DMI interface for efficiency reasons.

Before determining the internal structure of each module, we try to determine what kind of data and in what frequency and quantity this data flows from one component to the other, in order to choose the best implementation.

9.3.3 Data Flow between Tool and Server

Since CFI is a procedural interface, the data flow from the tool to the server and from the server to the tool consists of primitive C data types. So the granularity is also very small. It is not possible to control the granularity at this level, since CFI does not provide any specification about when the data should be transferred to the tool.

Possible scenarios:

The application programmer might decide to build a custom data structure in the tool, and then in one phase read in all the necessary data, perform the operations required by the tool. When that has been accomplished all the updated data might be returned to the CFI heap.
This implies that the CFI function calls occur in one burst, but the quantity of data per call remains constant. This is probably the case with simulator class tools (usually read-only) and batch tools such as in the case of flat extraction.

The application programmer might read in the data piece-meal, and write perform updates to the heap piece-meal. In this case the frequency of calls is evenly distributed. This is typical of editors.

The application programmer might not build his/her own data structures but constantly manipulate the objects in the CFI heap. This is characteristic of management functions, for example, changing an alternative.

### 9.3.4 Data flow between the DMI and the CFI manager

A question to be answered here is what data granularity is required from the DMI, should a complete View be read in or should lists be read in separately, etc. Figure 9.10 illustrates the two major domains. We see the following logical possibilities:

![Diagram showing Libdomain and Viewdomain](image)

**Figure 9.10. Lib and View domains.**

Read in all the data from view downwards. A View coincides with a NELSISS \textit{DO}. Therefore we can check-out/check-in a \textit{DO} and read in all the data. This solution is the simplest, since all the pointers between objects in the View domain can be relatively easily converted in
the same file. The drawback is that all the data must be put in the heap, while it is not clear that all the data will be needed.

Sub-divide the View domain into sub-domains of Netlist, InstanceList and PortList. This solution reduces the amount of data that needs to be put on the object heap, but requires a more sophisticated management since the pointers between the sub domains must be managed in the manner as the inter-domain pointers between View domains of figure 9.10.

9.3.5 Correlation between Object Types
It is reasonable that object types that are strongly correlated should be read-in together.

Scenarios

Editor scenario: A partially edited view is being manipulated. The next action is the addition of a Port. In normal cases the next action will be the addition of a Net. Since a Net must connect at least two terminals, it implies that the next action will be the addition of another Port, or a connection with a PortInstance. Since an Instance is associated with a PortInstance it must also be available. This scenario can be repeated for starting with a Net or a PortInstance. Thus the conclusion is that within a very short period all the four basic types manipulated in a View are necessary.

Simulation Scenario: A Net changes from low to high. The only way to determine what the effect of this change is on the status of other neighboring nets is to follow the connection of this net with others via the Ports and PortInstances. At the moment that the value of the signal on the Instances must be known these in turn must be made available (not always necessary to open the hierarchy because the instance might have a delay property).

From these two scenarios a strong correlation exists between Ports, Instances, Nets and PortInstances. A weaker correlation exists between the types and their Properties, since the Property is tool dependent. For instance, there is no good reason to read in graphical information of an object needed by an editor when performing a simulation.

The NetBundle instances could be excluded but this creates complex management problems because all the references from the Nets, the Port(Inst)Bundles and from the owning View to the NetBundle, have to be converted to pointers.

Conclusion
Everything in the domain of a View should be read-in together, excluding the properties.
9.4 The CFI Manager - Overview

We present a general overview of the manager (figure 9.11), based on chapter one. The CFI data manager is shown in somewhat more detail, namely it is divided into two modules, the CFI Data Repository Controller and the Object Table (OT) Manager.

![Diagram of CFI Manager](image)

**Figure 9.11. CFI manager.**

The data-flow between the several parts is represented with arrows. Each set of arrows is numbered for reference purposes. The following list summarizes the type of data.

Flow 1 & 2: Primitive data-types. Since the server part redirects the data, the data in point 1 will be the same manner as in point 2. Incoming data can consist of zero or more primitive data types and outcoming data consists of zero or one primitive data-type.

Flow 3: The data flow may consist of primitive CFI data types and internally used pointers. This flow requests the CDR Controller for a certain object or certain information of an object.
Flow 4: This flow may contain object ID’s and pointers to objects.

Flow 5: This flow can be used to get the pointer to a referenced object. In this case the CDR controller only knows the symbolic link to an object and not its pointer.

Flow 6: Flow between the CDR controller and the Object heap. This consists of object - ID’s, object pointers and attributes of these objects.

Flow 7: This flow consist of DMI structures (projkey, cellkey etc) and persistent data. Whenever an object is loaded into the heap, it is fetched from the database by the DMI interface, passed to the CDR controller via 7 and put into the CDR by the CDR controller. After these operations, the OT manager is updated via 5 and 6.

Flow 8: Direct flow between the framework and the CFI interface. Data same as 7.

9.4.1 The CFI Interface of the CFI Manager
The CFI interface consists of a fairly large set of functions. The procedural interface enables the manipulation of the various object types. Version 1.0 contains some 85 functions. Here follows an example (from V0.7).

```c
    cfdirViewIdT cfdirOpenView (libName, cellName, viewName, error)
    cfdirStringT libName;
    cfdirStringT cellName;
    cfdirStringT viewName;
    cfdirErrorT * error;
    {} 
```

9.4.2 CFI Data Repository

9.4.2.1 Data Structure Analysis
We choose to store the data managed by the manager on a volatile heap because this enables the fastest access to the data. (There is no requirement in the specifications that this data must be persistent.)

Based on the guidelines of the object types and their behavior as specified in the Express-G language, we create the data types in the form of objects. This seems to be the most natural way to uphold the interconnection between the data.

9.4.3 Graphical Representation of the Objects
We have defined all the objects graphically in order to highlight the interaction between objects. An example of the NetBundle object type is given below to illustrate the complexity of the inter-object relations.

9.4.3.1 C Structure Representation of the Objects
At this stage the data structure definitions can be made. Here we give one example, of the object type View. A View entity represents an abstraction of the design that may be instantiated
in another View to build a design hierarchy. It represents a specific implementation of a Cell. As such, a View is owned by exactly one Cell via the Owner attribute.
typedef struct _cfdrViewT{
    cfdrObjectTypeT ObjectType;
    cfdrObjectIdT ObjectId;
    cfdrPropsIdT Props;
    cfdrStringT Name;
    cfdrCellT Owner;
    cfdrPortsT Ports;
    cfdrNestsT Nets;
    cfdrInstsT Insts;
    cfdrViewTypeT ViewType;
    DM_CELL *cellkey;
}cfdrViewT;

Note the entity cellkey. This is a foreign attribute related to making objects persistent. It is a (pointer to a) NELSI S DMI library structure.

9.4.4 Discussion on the Object ID Problem

What is the best implementation of the object - ID mechanism? According to the documentation an Object - ID is an abstract data-type i.e. the framework supplier is free to choose it’s form. However, at the moment it is a 32 bit data structure. Furthermore, one of the functions puts a requirement concerning the implementation of this mechanism. This function may be called by the application without any preconditions. This implies that the object-heap manager at all times must be capable of answering the following question: "Is the object with the specified object - ID a valid object or not?"

Consider the following scenario: A tool has a pool of object ID’s which were returned by the CFI interface. Assume that one of the objects is deleted (indirectly) due to a CFI function call. Thus, an object ID in the core of the tool is not usable, but the tool is not aware of this.

Conclusions:

An exported object ID (an object ID that is returned to a tool) may be not recycled during the existence of the heap. This implies that an exported object ID cannot be a C pointer, since we do not have control over the OS allocation mechanism and it is not realistic to not allow freeing of object memory. Thus an object table is needed to hold the association between a exported object - ID
and the pointer to the object.

Since CFI does not permit direct access to objects on the heap the internal management may well (partially) be done using pointers as object - ID’s.

There are now two obvious possibilities:
Symbolic pointers.
All objects on the heap are handled as if they have been exported to the tool i.e. have exported object.
ID This means that the object management is uniform, but the runtime overhead could reduce performance by a factor of two.

Hybrid pointers.
Only exported object - ID's are put in the object table. Other object ID's that are (still) internal, remain pointers. A disadvantage is that a more complex management system is required.

![Diagram of Object Table and Heap]

Figure 9.13. CFI object-table.

We first examine the hybrid pointer implementation because it might be faster. Possibility B is oversimplified because it is not possible to implement all internal object - ID's as C pointers. We now analyze which pointers have to be symbolic.

9.4.5 Object Domains
From the CFI interface function definitions, two basic object domains can be seen as shown in figure 9.12.
These domains show the consequences for the internal referencing of objects, namely that references across domains require symbolic references, also for internal management as shown in figure 9.14.

Figure 9.15.1 is a conceptual illustration of the object heap. Two views are present. Object B is owned by object A, and is an instantiation (Inst or PortInst) of object C (View or Port). This instantiation relation is represented by the descriptor D.

Figure 9.15.2 shows how the descriptor pointer is implemented. It is a key in the object table (OT). This symbolic pointer is represented by a double-headed arrow. Its associated value is a C pointer to the object C.

In figure 9.15.3 we assume that the passivation of View 2 is occurring. The C pointer to object C is replaced by an object name (OName). We do this to remove the dangling reference to C and also to provide a handle on object C, which is not on the heap any longer, but still exists in persistent memory. From this state C might be activated (View 2 opened) or B might be passivated (View 1 purged). In the first case the object name is simply replaced by a C pointer to the re-allocated object, and the object - ID remains the same.

Figure 9.15.4 shows the second case. During the filing, when object B is filed the object name of C is fetched from the object table and stored in B's persistent image.
9.4.6 Counters
One of the problems is that any object which is a describer (View or Port) can never be removed from the object table, during the lifetime of the object heap. A way to solve this is to add a counter to the object table, only for describer types. If a key is given with the function `otmGetDescriptorOid()`, the counter is incremented.

9.4.6.1 The Internal Object-ID The internal object - ID is used internally by the CFI functions. These object - ID’s are simply C-pointers to structures and are not directly accessible to the tools themselves. These pointers are fast to access. In this case no conversions have to be made and immediate access to a object is possible.

9.4.6.2 The External Object ID
These object - ID’s are used by the tools and are returned by the CFI functions. The ID’s are also used to make cross references between different views. The ID’s are stored in one or more tables for fast access purposes.

Why use internal and external object - ID’s and not only C-pointers?
In certain situations C pointers cannot be used because the program or OS cannot or will not handle them. These situations lead to memory consistency violations.
9.4.6.3 The Object-ID Table
Different types of tables are possible. The two most obvious are:

An array containing all object-ID's. Since an object-ID is defined as a long integer it can be used as an index into the array. This has some major drawbacks:

Since all object-ID's are unique the array can grow to huge proportions. Some management is possible but very hard to implement.

The degree of usage can be very poor on larger projects

A hash table containing all object-ID's. They serve as a index in a table and whenever the index value exceeds the table size, its modulo is taken (index mod table-size). The table will not grow in length (longer). The object-ID can be generated in three different ways.

Just a counter. This way a unique ID is ensured and the overall degree of usage is very good.

A random generator which produces unique ID's.

A hash function for the name. The names have to be unique. This can cause some problems as the ID's have to be unique and duplicates can occur.

9.4.6.4 Hash Tables related to CFI Objects
Whenever an object is referenced by a CFI function, it is loaded into heap space or is supposed to be there. An object in heap space is merely a C structure and a object-ID to reference this structure. In order to obtain fast access to those structures, we might use one of the possibilities listed before. The most usable is probably the hash table.

An object can have two states:

The object is in the heap and is referenced by an instance. This reference is a hash table entry.

Now the instance can find the object in heap at all times.

The object has been removed from the heap (has been deleted or purged) and the hash table entry is cleared. Problems occur when the object is loaded into the heap space again. The object-ID must be the same otherwise all references from instances will point to the wrong object or to nothing at all. When we want to give the same object-ID again, we will have to remember the old object-ID.

Figure 9.16 is an example. It shows objects A and C and an instance B of object C. The object may be accessed either via the names table or the keys table. Initial measurements have shown this interface to be relatively fast. A rough estimation revealed that running the same tool with the OCT DBMS is two times slower.
Section 5. Measurements

9.5 Measurements

Some system performance measurements (implode and export) were taken based on an upgraded version of CFI (0.9.2), which does not differ essentially from the version described in this chapter (0.7). Both are implemented in C.

The test programs first created a heap of CFI library objects (such as Cells, Views and Ports etc) in working memory, then the implode and export times were measured. In this system the implode and export are not separate phases, but are executed sequentially per object or group of objects. The export time for each object was added up to obtain the total export time. The total implode time was obtained by subtracting the total export time from the save time of the total heap.

9.5.1 Test Results The results presented here are only an indication to the performance the interface. The measurements were set-up in a very simple manner, the number of tests performed was small and the work load of the computer was not completely stable. The measurements were performed at night to minimize interference. The tests ran on a SUN SPARC IPC with 24 Mega-byte working memory and 64 Mb swap space. The machine uses an external local disk (1.7 Gb of type Artacon). The measurements were performed on four different heap sizes. The biggest heap contained 5500 objects and the smallest heap contained 2000 objects.
Figure 9.17 shows the implode and the export times as a function of the number of objects in the heap. On average, implode took 2.0 milliseconds per object and export took 0.72 milliseconds per object. The objects in the heap were representative of an actual electronic design, and included all CFI object types. The implementation differs from the design, since currently all inter-object relations are maintained via the object table, which is the structure depicted in figure 4.3 plate C. This has consequences for the run-time speed.

![Diagram showing the relationship between the number of objects and time for implode, export, and implode+export operations. The x-axis represents the number of objects and the y-axis represents time in seconds.]

**Figure 9.17.** Implode and export measurements.
Chapter 10

A VOIS Based CFI Interface - Case Study

10.1 Chapter Overview

The first part of this chapter we investigate the interfacing of a CFI-DRPI with OPI.

The study of the VOIS implementation of CFI demonstrates the power of such a system to deal with persistency problems. The approach is different, since the interface designer may proceed from the point of view that the design objects are virtual. The designer builds the application on the OPI system, building on the base object opiVirtual. First the code is compiled with the system in non-persistent mode, which disengages the persistent functionality of the base object. When satisfied with the operations, the implementor may switch to persistent mode and incrementally introduce persistency aspects.

In second part of this chapter we present preliminary measurements on the performance of OPI.

10.2 The Incremental Design Phases

10.2.1 Step 1 - Volatile Functionality

The functionality of the application is studied and the necessary classes are defined. In the case of CFI this is straightforward, since the definition is given in an object oriented manner, as well as the hierarchical relationships as expressed in the Express-G net-list connectivity model. The base object of CFI, cfidObject is implemented on top of the opiVirtual class so that all the CFI objects inherit the virtual aspects, however at this stage the application is compiled without any persistency initialization. This disengages the persistent characteristics of opiVirtual and the objects behave like regular volatile objects.

As an example, figure 10.1 shows a (C++) class definition of a CFI type.
class cfidrView; public cfidrNamedObject
{
public:
  cfidrView (cfidrCell * , cfidrStringT, cfidrViewTypeT);
  virtual ~cfidrView ();
  cfidrViewTypeT cfidrGetViewType();
  void SetViewType(cfidrViewTypeT);
  virtual classType isA () const;
  virtual char * nameOf () const;
  virtual hashValueType hashValue () const;
  virtual int isEqual ( const Object& ) const;
  virtual void printOn ( ostream& ) const;
  virtual void opiPrintOffsets();

  cfidrNetScalar* cfidrCreateNetScalar(cfidrStringT);
  cfidrIter* cfidrGetNets(cfidrIterModeT mode);
  cfidrIter* cfidrGetPorts(cfidrIterModeT mode);
  cfidrIter* cfidrGetInsts();
  cfidrPortScalar* cfidrCreatePortScalar(cfidrStringT, cfidrPortDirectionT);
  cfidrInst* cfidrCreateInst(cfidrView *, cfidrStringT);
  //cfidrNetBundle* cfidrCreateNetBundle(cfidrStringT, cfidrStringT);
  //cfidrPortBundle* cfidrCreatePortBundle(cfidrStringT, cfidrStringT, cfidrPortDirectionT);

private:
  cfidrCell * Owner;
  cfidrList * Ports;
  cfidrList * Nets;
  cfidrList * Insts;
  cfidrViewTypeT ViewType;
};

Figure 10.1. CFI class definition.

There is no hint of persistency in the class definitions and the data generated will only survive
the life-time of the process.

10.2.2 Step 2 - Primitive Persistence
The source file is compiled with persistency initialization. Now the virtual functionality will
automatically be present in all the CFI class types, and a default DBMS is supplied by the
system. Figure 10.2 is an example of an application:
int main () {
    cfidrLib * lib;
    cfidrCell * cell1, * cell2;
    cfidrView * view1, * view2;
    opiVirtual * db;

    opiTransaction(INIT);
    db = opiInitializeDb(CREATE|OPEN);
    lib = new cfidrLib("FirstLib");
    cell1 = lib -> cfidrCreateCell("FirstCell");
    cell2 = lib -> cfidrCreateCell("SecondCell");
    view1 = cell1 -> cfidrCreateView("FirstView", CFIDR_NETLIST_VIEW);
    view2 = cell2 -> cfidrCreateView("SecondView", CFIDR_NETLIST_VIEW);
    //object manipulation
    .
    .
    opiTransaction(SAVE&Quit);
}

Figure 10.2. Application program example.

At a later stage the database functions may be integrated into the CFI top level objects, e.g. in the constructor of cfidrLib, thus removing the need to explicitly add code to the main program. Suppose that more than one database is desirable. In this case the programmer must associate a top level object with the corresponding database, for example, figure 10.3:
int main () {
    opiDb * db1;
    opiDb * db2;
    cfidrLib * lib1, lib2;
    cfidrCell * cell1, * cell2;
    cfidrView * view1, * view2;

    opiTransaction(INIT);
    db1 = opiInitializeDB(CREATE&OPEN, "DB1");
    db2 = opiInitializeDB(CREATE&OPEN, "DB2");
    lib1 = new cfidrLib("FirstLib");
    lib1-> opiSetSystemPtr(db1);
    lib2 = new cfidrLib("SecondLib");
    lib1-> opiSetSystemPtr(db2);
    cell1 = lib -> cfidrCreateCell ("FirstCell");
    cell2 = lib -> cfidrCreateCell ("SecondCell");
    view1 = cell1 -> cfidrCreateView ("FirstView", CFIDR_NETLIST_VIEW);
    view2 = cell2 -> cfidrCreateView ("SecondView", CFIDR_NETLIST_VIEW);

    // Object manipulation
    
    
    opiTransaction(SAVE&QUIT);

    Figure 10.3. Multi-database instances.

If there are inter-object relations between the two databases, the system will perform a partitioning anyway, however there is no guarantee that the partition will be according to the programmer’s wishes. This requires additional configuration data.

10.2.3 Step 3 - Primary Clustering
First the first-order configuration data is defined. This involves the definition or extension of the transform function and relations in the configuration file. The relations involve defining the main clusters. In this case, at the cfdrlLib level and the cfidrView level, and making modifications to the transform and transport capabilities if necessary. The configuration file of figure 10.4 gives an example of primary clustering.
CLASS Lib DBCLUSTER PRIVATE
CLASS Cell CLUSTER GENERAL CHILD OF Lib
   ATTRIBUTE Owner PERSISTENT POINTER TO Lib
CLASS View DOCLUSTER GENERAL CHILD OF Cell
   ATTRIBUTE PortBundle PERSISTENT POINTER TO Ports
   ATTRIBUTE ViewType PERSISTENT

Figure 10.4. Configuration data example.

10.2.4 Step 4 - Secondary Clustering
Second-order configuration involves secondary clustering if desired and possible reimplementation of the transform/transport functions (figure 10.5).

CLASS Lib CLUSTER PRIVATE
CLASS Cell CLUSTER GENERAL CHILD OF Lib
   ATTRIBUTE Owner PERSISTENT POINTER TO Lib
CLASS View DOCLUSTER GENERAL CHILD OF Cell
   ATTRIBUTE Owner PERSISTENT POINTER TO Cell
   ATTRIBUTE PortBundle PERSISTENT POINTER TO Ports
   ATTRIBUTE ViewType PERSISTENT
CLASS Ports CLUSTER GENERAL CHILD OF View
CLASS Nets CLUSTER GENERAL CHILD OF View
CLASS Insts CLUSTER GENERAL CHILD OF View
TRANSPORT opiBinary
TRANSFORM opiBinary

Figure 10.5. Second-order configuration definition.

In this configuration file, second-order partitioning has been performed on Ports, Nets and Insts. The tool may be re-profiled, and the speed and resource performance may be compared with the primary profiling. The programmer may change the configuration without any recoding of the tool. Also here the programmer has defined his own transform and transport functions, of type binary to improve the run-time I/O.

10.2.5 Step 5 - Partitioning Acceleration
Assuming that the tool has been stabilized and has been profiled, the partitioning may form a bottleneck. In this case relations may be hard-wired in the code using the opiSetSystemPointer function, as shown in figure 10.6. This is also done in figure 10.3 to make the link between the database and the library objects.
.cfdirLib * lib = new cfdirLib("FirstLib");
.cfdirNetScalar * net2 = new cfdirNetScalar (view2, "net2");
.cfdirPortScalar * port21 = new cfdirPortScalar (view2, "port21",
    CFDR_INPUT);
.cfdirPortScalar * port22 = new cfdirPortScalar (view2, "port22",
    CFDR_INPUT);
.cfdirProp * prop1 = new cfdirProp ("prop1");
.cfdirProp * prop2 = new cfdirProp ("prop2");
/* hard-wire certain relations in the heap structure */
lib -> opiSetSystemPointer (_opi UniverseRoot);
cell1 -> opiSetSystemPointer (lib);
cell2 -> opiSetSystemPointer (lib);

Figure 10.6. Acceleration through hard-wiring.

10.3 An Alternative Virtual Realization of CFI DRPI

A more generic object oriented approach is not to directly implement the CFI classes and
operations but first to construct a tool kit of objects that possess the characteristics necessary for
upholding the CFI functionality. It is best to utilize the inheritance mechanism available to the
programmer. Although supported by some C++ compilers, in the present CPI implementation
the impact of multiple inheritance has not been studied. Therefore objects with more than one
of the characteristics listed below cannot be implemented using this approach. The important
aspects in CFI are:

Owner which implies that the object has responsibilities pertaining to certain objects it
references and Owned which implies that an object is aware that it is tightly related to a
particular object.

Bundler which implies that the object contains a particular type of collection and Bundled
which implies that the object forms part of a bundle.

Describer implies that the object forms the template for another object the described object.
Therefore the described object need only be a reference to the describer, rather than being an
exact copy of the describer.

List collection: any object has the capability to contain a list of lists. Any object may contain at
least one list, known as the property list, since this may occur in any object.
10.4 Analysis of a Simple Electronic Component

The goal of this section is to illustrate various potential inter-object relation problems that may occur during design, and to explain how a VOIS, such as OPI, can overcome these problems. We shall utilize the structure of a 4-bit register, constructed internally of 4 flipflops. Each flipflop contains 2 NAND gates. First a hierarchical decomposition of the component is presented with the help of several drawings. Thereafter, we shall show how this component is mapped to the CFI (Version 0.7) [17] circuit design representation format. We use the partitioning configuration defined in the previous section. We shall focus on the problematic relations that are considered to be complex in conventional systems.
The circuit example shown in figure 10.7 is a four-bit data latch (quadlat). The left and right hand columns present the internal and external representation of a component respectively. The functional behavior of the component is as follows: A 0 pulse on the reset line brings all 4 output lines (d_out 0:3) in a 0 state. If one of the latches detects a 0 on its input data line (d_in 0:3) the corresponding output line switches to 1, and remains there until a reset occurs. The circuit is constructed of 2-input nands (nan2). Two nands form a set-reset flip-flop (ff) and four flipflops compose the quad-latch.

10.4.1 Component Identification

In figure 10.8 we present a notation to enable identifying object instances of a particular electronic type. For example, the lowest nand port is a CFI net-list type Inst called nan22, of a Net-list type View, called nand2, which itself is instantiated within a Net-list type View called ff. A second example; rx1 is a Net-list type Net, that connects Net-list type Port r with Net-list type PortInst and x1->nan21.

We shall now map the example circuit to CFI Net-list components. According to the CFI conventions, the electronic types are defined as the name of a generic Net-list type. (See figure 7.9 for a legend).

10.4.2 Top Hierarchy

Let's assume that the component is part of the net-list type Library (figure 10.9), called logic. Logic contains three cells, quadlat, ff and nan2. Each cell contains at least one View instance. A Cell (ff) instance might contain several View instances, for example ff might have two Views, one typical (typ) and one slow (slow).
Section 4. Analysis of a Simple Electronic Component

10.4.3 Instance Hierarchy
Figure 10.10 shows how Views are instantiated in other Views. For example, Cell ff View typical is instantiated four times (ff1...ff4) in Cell quadlat View typical.

10.4.4 Fine-Grained Components
In figure 10.11 the nets, ports and port-instances have been added. This is a complete representation of a flip-flop. Properties have been neglected.
Figure 10.10. View instantiation.

From the view point of an interface builder a number of troublesome relations are visible. In conventional interfaces the designer must determine the physical distribution of data in advance. This is the norm since in the CFI specifications attention is paid to what objects are expected to be present, given the presence of another object. Take the following postcondition associated with cfdrOpenLib function: "Opening a Library means the object hierarchy owned by the Library is made available until Views are encountered. Views are opened independently via cfdrOpenView". A postcondition associated with cfdrOpenView is "Opening a View means the object hierarchy owned by the View is made available".

These guidelines are based on the fact that conventional interfaces have problems dealing with data partitioning, since this isn’t automated, as shown in figure 10.11. Thus, objects above the horizontal dashed line, *lib logic*, cells *ff* and *nan2* should be accessible. With a conventional interface, this means that the cells are actually loaded. On the other hand a *vois* could load the cells on demand if so required.

Often, several hundred cells may be created. The problem becomes more acute when several Views must be made available, for example, in the case of *Cell* instantiation. Actually only the
instances belonging to the Cell are required, i.e. in figure 10.11 only nan21 and nan22 are required, and in a conventional system all other objects (ports, nets and port-instances) in the bottom partition might be loaded.

![Diagram of a flip-flop](image)

**Figure 10.11. A complete flip-flop.**

Apart from being difficult to implement, hard-wiring the partitioning in the code implies that any partitioning changes are extremely costly. For example, suppose a more detailed partitioning were desirable for performance reasons (as shown by the dotted lines in figure 10.11). At instance level, an example of a partitioning problem is the relation between the Net rX1 and its View. If the data are not stored contiguously, the bidirectional pointer must be reconstructed, and a dedicated function is necessary to load rX1’s partition from the disk. Since the loading is driven by specific functions, the number of functions increases. Another example is the description of the function cfidrGetOwner, which specifies "The Owner relationship is defined when an object is created and cannot be modified". This constraint is mainly due to the manner in which partitions are constructed, i.e. it is difficult to transfer an object from one container to another. In a VOIS this would automatically cause a (hierarchical) migration. A further example is the descriptor relation.
Figure 10.12. Net and Port bundles.

The describer of an Inst is a View instance in another Cell instance. For example, the pointer between nan21 to the View type circuit of Cell nan2. According to the CFI specifications, before attempting to access the describer object, one must first check whether it is available. The function cfdcrCheckDescriptor specifies that "If the Descriptor View isn't opened yet, then this function will open it". This problem is also reflected between the Port describer of a PortInst, e.g. the PortInsts of the Inst nan22 refer to the Ports x1, x2 and y in the View type circuit of Cell nan2. This is a consequence of the partitioning between different cells (views). The function cfdcrDeleteObject is problematic. A condition stated is "Destroying an object doesn't destroy the persistent data associated with that object, the update caused by the destruction must be saved in order for the persistent data to be destroyed". This requirement is inherent in a VOIS. Yet another example: assume that the View typ of Cell nan2 has to be deleted. According to the CFI specification, a delete is hierarchical. Therefore ports x1, x2 and y must be removed. In a classical interface, implementing such an operation is complex. In a hierarchical VOIS hierarchical traverse facilities are inherent in the system. Another aspect is the describer relation leading to the deleted Ports and View, from nan21, nan22 and their PortInsts. In a conventional system this can lead to dangling references, which can cause the
tool to crash. In any case, the consistency check is complex. In a VOIS a delete could be regarded as a migrate to a special cluster instance called Delete cluster. When a save is performed, the partitioning mechanism can detect any references to deleted objects, since these all have a logical-ID beginning with the ID of the Delete cluster. This approach also enables an Undelete operation, since the deleted object hierarchy still exists.

10.4.4.1 Bundles
Complexity increases if also Net and Port bundles are supported as depicted in figure 10.12. For example, suppose we want to move the Port input_0 out of the PortBundle input_0:4. According to CFI, ownership relations mustn’t be changed, therefore the only way is to destroy the entire PortBundle, and then to recreate input_0 (it was destroyed too). However, all the inputs are connected to Nets that are in NetBundle, therefore, all the Nets must be destroyed, NetBundle must be destroyed and everything recreated. This complex operation must be coded in the tool. In a VOIS, changing the Net input_0s ownership to the Net will be automatically tracked through the migration system.

10.5 Preliminary Results

This section presents preliminary findings based on the CFI system coupled with the C++ CFI-interface presented in this chapter. The system has not yet been optimized or tuned. The tests were run on a HP 9000 series 300 work-station computer (MC68030, 50 mega-hertz) with 8 mega-byte working-memory, and 64 mega-byte swap space. Due to time limitations, the C++ CFI-interface only included elementary types, (i.e. it did not include any bundle types). The tests were simple and were not carried out a sufficient number of times to be considered accurate, and there was little control over the computer and network environment. The tests were run at night to minimize the interference. The results are given to provide an indication to the behavior of the system. Detailed measurements and an analysis are ongoing and are to appear in [54].

10.5.1 Test #1: Totally unpartitioned heap (partitioning and implode)

In this test a heap of CFI objects was created, partitioned and then imploded. The configuration data was complete (it included information on attributes). The test was executed for a varying number of objects, as depicted in figure 10.13. The graph shows the partition and implode times, versus the total number of (virtual) objects in the heap. The dashed line represents the implode time. The full line represents the partitioning time using the MANY_UNPARTITIONED algorithm. The dotted line represents the partitioning time using the FEW_UNPARTITIONED algorithm. The maximum number of objects was restricted to 6000 to eliminate interference caused by swapping (for visibility, only 3 measuring points are shown, but a measurement was made every 400 objects). Both the implode and the partitioning appear to be linear.
all objects in the heap were unpartitioned
configuration data was complete

Figure 10.13. Basic partitioning and implode measurements

10.5.2 Test #2: Partially unpartitioned heap, MANY_UNPARTITIONED algorithm

In this test a heap of CFI objects was created and partitioned. Thereafter, new objects were created and the heap was again partitioned. The objective was to see the behavior of the MANY_UNPARTITIONED algorithm under the initial condition that the heap was partially partitioned. Measurements were done for 1000 up to 6000 objects. For each case, varying ratios of partitioned objects vs. unpartitioned were created, starting from 17% unpartitioned, up to 85% unpartitioned. In figure 10.14, the partition time is given for a varying total number of objects. The slight deviations with respect to test 1. In various situations, a large percent of the objects in the heap may already be partitioned, in which case the partitioning time is much better. For example, in the case of 17% unpartitioned objects, the partitioning is 3 times faster than in the case of 85% unpartitioned objects.
algorithm MANY_UNPARTITIONED was used

data configuration was complete

[seconds]

85%  1.2 [ms/object]

68%  1.0 [ms/object]

51%  0.8 [ms/object]

34%  0.6 [ms/object]

17%  0.4 [ms/object]

[#objects]

Figure 10.14. Partial partitioning measurement (MANY_UNPARTITIONED algorithm).

10.5.3 Test #3: Partially unpartitioned heap, FEW_UNPARTITIONED algorithm

This test is similar to test 2 with the difference that the FEW_UNPARTITIONED algorithm was used. Figure 10.15 shows the measurements. For 17% unpartitioned objects, FEW_UNPARTITIONED is faster than MANY_UNPARTITIONED (see test 2) by a factor of 2. For 85% unpartitioned objects, FEW_UNPARTITIONED is just as fast as MANY_UNPARTITIONED. This indicates that the FEW_UNPARTITIONED algorithm can probably always be applied.
algorithm FEW_UNPARTITIONED is used
data configuration is complete

![Graph showing partial partitioning measurement](image)

**Figure 10.15.** Partial partitioning measurement (FEW_UNPARTITIONED algorithm).

### 10.5.4 Test #4: Best case test for FEW_UNPARTITIONED

A heap of 3000 objects was created and then partitioned. Three new objects were then created and partitioned using the FEW_UNPARTITIONED algorithm. The configuration data of the objects was specially set up so that each object could be partitioned by traversing only one cluster. With FEW_UNPARTITIONED, the objects (all belonging to the same cluster) were partitioned within 15 milli-seconds, while MANY_UNPARTITIONED took 820 milli-seconds to partition the heap, i.e., a speed up factor of 50.

### 10.5.5 Test #4: Worst case test for FEW_UNPARTITIONED

A heap of 800 objects was created and then partitioned. One new object was then created and partitioned using the FEW_UNPARTITIONED algorithm. The configuration data did not give any
information regarding which cluster class to search, so FEW\_UNPARTITIONED had to traverse the whole heap. In this case FEW\_UNPARTITIONED performed slightly slower than MANY\_UNPARTITIONED. FEW\_UNPARTITIONED took 210 milli-seconds to partition the heap while MANY\_UNPARTITIONED took 200 milli-seconds.

10.5.6 Other results

Currently, the (C++) CFI interface using OPI requires 71 kilo-bytes when loaded into memory. The disk image is 446 kilo-bytes. This executable includes the NELSIIS interface library, the OPI libraries and the C++ CFI interface library (the latter is a simple version). The shared memory and inter-tool communication modules are not included.

It is difficult to compare the performance of the two interfaces due to the difference in hardware configuration, and the fact that the C CFI interface currently manages more complex relationships (bundles). The C CFI interface measurements were taken on a SPARC IPC\(\text{SUN4C}\) estimated to perform 14 mips while the C++ CFI interface using OPI ran on a HP-300 estimated to perform 10 mips. The implode for the C interface was estimated at 2.0 milli seconds per object (section 9.5). The estimation for the C++ interface was also 2.0 milli-seconds. The export time for OPI has not been measured but it is probably similar to the C interface (0.7 milli-seconds per object). The partitioning time for the C++ interface varies between (worse case) 1.2 to (best case) 0 milli-seconds depending on the state of the object heap, the quality of the configuration data and whether any "hard-wiring" has been done in the code. Suppose we estimate an average of 0.6 milli-seconds to partition an object, then saving an object with the C++ interface is 3.3 milli-seconds per object, while saving an object with the C interface is 2.7 milli-seconds. In OPI the loading of objects occurs on demand therefore we do not expect this to degrade run-time performance. The speed of executing application methods is probably close to a normal C++ application, since inter-object referencing is done with C pointers. If these assumptions are correct then the difference between the C and the C++ interfaces is the partitioning overhead which on average could add an additional 22% overhead to the save.
Chapter 11

Conclusion

11.1 Experience and Problems

We believe the dependence of a VOIS on an object-oriented language is acceptable since the language is also effective for implementing large software systems. Keeping the procedural interfacing code out of the client program and the definition of the application classes is possible but should not be done on principle but depending on the situation. Efficiency can be improved considerably by making the distinction between which types are persistent and which aren’t, knowledge which is usually available to the designers of hierarchically structured systems. Furthermore, in the spirit of C++, the user can reduce transparency to enhance performance. Persistent information can be "wired" into the application classes. The manner in which objects are persistently formatted can be changed by overriding the default implode scheme. The level of transparency is also dependent on the usage of the system. If used as the basis for a higher level management system, then it is wise to make optimal use of the management capability and information that already exists in the system and to tap into it. This requires a greater understanding of the system. For example, bi-directional management of the owner object is present via the system pointer and therefore defining an extra owner attribute is transparent but redundant. The problems encountered in this work may be categorized as follows:

Deficiency in the object-oriented language

C based OOLs that support backward compatibility with C are difficult to model due to the number of exceptions to the rule which must be studied and accounted for. The flexibility of C is, to an extent, contradictory to the object-oriented paradigm. This has the effect of cluttering up the picture to the extent that the object-oriented concepts are no longer clear. The choice for implementing OPT in C++ was mainly based on the fact that it is (today) the commercial standard, readily available on most work-stations and has the fastest run-time speed. However, if one could put aside environmental influences, the author's personal choice of classical object-oriented language (with respect to languages he has extensively used) as a programming medium would be Smalltalk, Objective-C and C++, in that order of preference. This order coincides with how true the language is to the concepts of object-oriented programming.

Lack of experience in object-oriented programming

Good object-oriented programming is an art in itself, which is slowly and painfully being
consolidated [76]. The author’s experience is that bad object-oriented programming causes more problems than bad procedural programming. The author’s inexperience in the usage of object-oriented languages at the initial stage of development took its toll in terms of implementation effort.

11.1.1 VOIS requirements from an object-oriented language

Based on the OPI experience, the author finds the following properties to be desirable in an object-oriented language.

Inheritance support
Application class types must inherit the fundamental virtual characteristics which are upheld through the inheritance mechanism. (If a language without built-in inheritance is used, then of course this inheritance will be implemented at one’s own cost). Design and implementation of good inheritance mechanisms is a specialized field and if available should be used.

Triggering support
The trigger enables transparently performing system related activities before or after performing the intended application method. Triggering the persistent activities is a sub-set of the more general need to trigger any kind of event. If this is not supported by the language, a parser is needed to automate the insertion of such a trigger into the code. This increases the visibility of the virtual mechanism to the programmer which is undesirable.

Sender’s address support
When an object is receives a message, it should be possible to derive which object sent the message. In the virtual paradigm, this enables an object to determine whether it should associate itself with the sender or not. This functionality can only be satisfactorily supplied by the compiler vendor. Currently, object-oriented programming is centered around the capacity of an object to provide a service. A shift toward object interaction is required, since performing a method is a relationship with the attributes client, server, and request. This concept is supported at class level in C++. A method of a class may be classified to provide service to a particular category of classes.

System Pointer support
Static information is commonly supported in object-oriented languages with the *is_a* variable which gives the object knowledge of its class. Dynamic information is commonly not supported. The System Pointer gives the object knowledge of its run-time environment.

Polymorphism support
This implies using a generic *ld* type for defining an object variable, regardless of its class (Objective-C, Smalltalk). Although better type checking may be performed at compile time, defining an object variable as a (pointer to a) class type (C++), causes technical difficulties when implementing a VOIS.
The cost of automation:
Automating persistency implies replacing dedicated (and usually efficient) code written by the
programmer by a management component. This inevitably requires more computation
resources; however, the clustering, implosion and explosion overhead might be compensated by
the look-ahead potential, partly present in the OPI architecture. Obtaining this information
requires extensive evaluation with a mature system of tools running on the VOIS. At this
prototypical stage it is too early to draw conclusions.

11.2 State of the Art VOISs

When the author began this work, the field of VOISs was relatively barren. One of the main
reasons was probably the caution regarding object-oriented languages and programming, and
the fact that the basic DBMS problems were using up most of the resources. However in the last
five years, the object-oriented paradigm has changed from a theoretical research topic to a
practical commercial product and has penetrated almost all commercial software areas. Many
object-oriented languages have been developed, and some will become standard packages of
operating systems. The backing of the industry has caused a huge acceleration in the field of
object-oriented methodology and technology. Two representative object management systems
that are C++ based are Objectivity and Onios, which were shortly introduced in chapter 2. An
extensive description of various systems and basic differences may be found in [77].

11.3 The Future of Interfacing

11.3.1 Research progress in programming languages.

Compiler-interpreter hybrids
Hybrids such as Self [78] are an important advance in object-oriented technology, since object
classes can be created on the fly. Self is UNIX-based and has a reasonable run-time performance.
It is a hybrid between an interpreter and a compiler based language. Actually "new" code is
compiled on the fly and saved in the so-called virtual machine. Thus, it provides the capability
of a Smalltalk interpreter with the speed of an Objective-C compiled program. The importance
of this technology is that compiler based languages have always limited data-structuring for
generic objects. For example, the representation of a flipflop defined by an end-user is in CFI
terms, a View type instance, whose name attribute is flipflop, and an instance of a flipflop is an
Inst type instance whose describer is the flipflop instance of type View. This definition is
caused by the limitations of compiler based languages. However, at a conceptual level, flipflop
can better be viewed as a specialization of a View of type FlipFlop. Then, an instance flipflop
is an instance of type FlipFlop. It should also be possible to add additional functionality or
attributes associated typically with flipflops. Self has broken through this barrier. Currently,
Self does not have built-in persistency (queries into this possibility have appeared in the Self
group e-mail bulletin, dated September 1992), however, if the object type data stored by the virtual-machine were to be stored in a DBMS, the possibility of dynamically creating object types and storing them, rather than just instances of pre-compiled types is an important step forward.

**Parallel languages**
Parallel programming has an important consequence for the speed of virtual object systems. Current VOIS systems are slower than such approaches as the NELTIS DML, since the operations on a particular machine are sequential. The number of computations involved in exploding and imploding an object in an automated way is invariably larger than when explicit commands are coded in the program. However, a hierarchically structured VOIS has a natural capability to perform anticipation as to which clusters might be required next. Provided that sufficient memory resources exist, these may then be loaded in advance. On a parallel machine this could be performed without loss of tool power. The look ahead mechanism may also be effective on a sequential machine, for the category of interactive tools, e.g. editors. These are usually idle, waiting for the user to make a request. Suppose the user prompts for loading in a hierarchical cell; the waiting time for the user is often frustrating. With the lookahead mechanism, the cell might already be loaded, and could be made available in a fraction of a second, since the only operation necessary would be notifying the DBMS that a potential check-out has become actual.

**11.3.2 Operating system evolution.**
A new generation of object-oriented operating systems is dawning. One is already commercially available [79]. NeXT is a commercial work-station that uses an object-oriented language for implementation of the operating system. This creates a much tighter binding between the disk and the programming interface. For example, in the conventional UNIX operating system, and C programming language setup, calling most of the operating system functions from within a program is not compatible with function calls from a C library, but much more restrictive and slow. This forms a clear interfacing barrier for the programmer. The NeXT operating system is based on Objective-C and components in the system are considered objects, which may be manipulated using an Objective-C library. This creates new possibilities for the implementation of persistency.

**11.3.3 Future Work**
**Short term**
Although the concepts of object-oriented programming and persistency are now household words, the realizations of key subsystems such as (hierarchical) complex objects, relative addressing methods, transformation and transportation have not yet been consolidated. A serious evaluation of several systems is required; at the architectural level by comparing schemas and reasoning about the relationships and functionality, and at the operational level by benchmarking. This should provide developers of VOIS with guidelines on the best mechanisms, or at least the pros and cons, in order to perform a good selection.
**Future of CAD Interfacing**

It seems that there are two possibilities, the first is building general object-oriented functionality into the operating system, which will encourage the development of object-oriented and persistent programming languages. The other is supplying an accepted DBMS library together with the operating system as a package deal, with slight enhancements of the operating system, and upgrading of standard object-oriented languages with persistent functionality.
Appendix

OPI Information Schema

Figure Appendix 1. The explicit schema
Figure Appendix.2. The implicit schema
References


Samenvatting

Software systemen die veel taken uitvoeren hebben een complexe structuur. Ontwerpers van zulke software systemen beheersen de complexiteit door het systeem uit diverse onderdelen op te bouwen. Elk deel - module - voert een specifieke taak uit.

Een programma is een software module, die de gebruiker van een software systeem in staat stelt een taak uit te voeren. Het aantal programma's in het systeem neemt toe met de diversiteit aan taken die de gebruiker uitvoert. Gegevens worden gemeenschappelijk genoemd indien ze zijn gecreëerd met één programma om gebruikt te worden in andere programma's. Persistente gegevens zijn die gegevens die blijven bestaan nadat het programma, waarmee die gegevens zijn gecreëerd, is beëindigd.

Een database-module wordt op grote schaal in systemen gebruikt om de gemeenschappelijke en persistente gegevens te beheren. De taak van de database is het coördineren van het gebruik door de programma's van de gemeenschappelijke gegevens en om persistente gegevens naar een opslaggeenheid over te brengen.

De modulaire structuur van het systeem vereist dat de modules onderling communiceren. Een programmeer-interface is een speciale module die een programmeur in staat stelt deze communicatie te beheersen. Toen begonnen werd aan dit onderzoek was de gebruikelijke methode van interfacing een procedureele. Een procedurele interface bestaat uit een verzameling procedures en structuren die de overdracht van gegevens tussen programma's en database mogelijk maakt.

Bij procedurele interfacing treden de volgende problemen op:
(i) De programmeur moet de procedures en de structuur van de gegevens leren;
(ii) De programmeur moet de procedures inbouwen in het programma om het gegevens transport tussen programma en database te beheersen;
(iii) De gegevens structuur die ontstaat bij het gebruik van het programma verschilt vaak van de structuur die door de database gebruikt wordt.

Deze complicaties motiveerde de verleden onderzoekers om een programmeer-interface te ontwikkelen die voor de programmeur eenwoudiger te gebruiken is. Dit probleem is in dit proefschrift aangepakt door ervoor te zorgen dat objecten virtueel lijken voor de programmeur. Een virtueel object bestaat in de geheugenruimte van een programma op het moment dat het nodig is. Als de interface het gebruik van virtuele objecten ondersteunt hoeft de programmeur niet het transport van gegevens tussen programma en database te programmeren en is er dus geen noodzaak om deze procedures in het programma op te nemen.

De term Virtual Object Interface System (VOIS) wordt gebruikt om een interface te beschrijven dat gebruik maakt van virtuele objecten. In dit proefschrift worden de concepten van virtual
object interfacing beschreven en worden verschillende problemen en oplossingen behandeld gebaseerd op een experimentele VOIS, genaamd OPI. Ook wordt het gebruik van het systeem onderzocht om het gegevensbeheer en de communicatie tussen programma’s te verbeteren. De nadruk ligt hier op de interfacing van elektronische ontwerp programma’s die gebruikt worden bij het ontwerpen van compacte elektronische schakelingen.

OPI is geschreven in een object-georiënteerde programmeertaal. Bij het gebruik van zo’n taal beschouwt de programmeur de gegevens in het programma als een groep interactieve objecten. De objecten sturen elkaar requests om diverse taken uit tevoeren. Inheritance en encapsulation zijn belangrijke, door de object-georiënteerde taal ondersteunde, eigenschappen voor de implementatie van virtuele objecten. Inheritance stelt een object-type in staat om zowel gedrag als structuur van gegevens over te nemen van een ander object-type. In OPI bestaat een objecttype dat virtuele eigenschappen heeft. Door de inheritance-eigenschap kunnen objecten die door de programmeur ontwikkeld zijn die eigenschappen erven. Encapsulation betekent dat de interne gegevens van een object niet direct toegankelijk is voor andere objecten; zij moeten een request zenden naar het object. Encapsulation staat objecten toe in de geheugenruimte van het uitvoerende programma te bestaan zonder de interne gegevens die in deze objecten staan werkelijk uit de database te laden. De gegevens worden alleen in het object geladen als het een request ontvangt.

In hoofdstuk 1 wordt het interfacing probleem en de traditionele oplossing op het gebied van elektronisch ontwerpen geïntroduceerd. De software modules die gebruik maken van de interface worden besproken. Ook worden de karakteristieken van verschillende interface-systemen gepresenteerd. In hoofdstuk 2 wordt de automatisering van interfacing processen behandeld. De wijze van gegevens overdracht wordt geanalyseerd en definities van de te automatiseren handelingen worden gegeven. De keuze van een object-georiënteerde programmeertaal voor de implementatie van het interface-systeem wordt gemotiveerd en verschillende andere interface-systemen worden gepresenteerd. In hoofdstuk 3 komen de management modules, die nodig zijn voor de interfacing in OPI, aan de orde. Vervolgens wordt een overzicht van de run-time omgeving en de inpassing van OPI in een programmeer-omgeving getoond. Hoofdstuk 4 laat de eigenschappen zien van het virtuele object en hoe deze eigenschappen geïmplementeerd zijn. Hoofdstuk 5 handelt over de groepering van objecten die bij elkaar opgeslagen moeten worden en hoe de programmeur dit kan beheersen. Partitionerings-algoritmes worden gegeven. De eerste resultaten wijzen erop dat de partitioneringstijd lineair blijft met het aantal objecten. In hoofdstuk 6 wordt beschreven hoe de OPI kernel uitgebreid kan worden om het ontwerpmanagement van kleine objecten te ondersteunen. In hoofdstuk 7 wordt een voorstel gedaan hoe de VOIS kernel gebruikt moet worden ter ondersteuning van concurrent data sharing tussen programma’s. In hoofdstuk 8 wordt een test systeem presenteerde. De programmeeromgeving moet een testsysteem hebben om programma’s te testen die in ontwikkeling zijn. De gekozen benadering is om de test te beschouwen als een object, en vervolgens een object-georiënteerde database te gebruiken om de tests te beheren. Hoofdstuk 9 behandelt een praktijk voorbeeld van een procedurele
interface, genaamd CFI-DRPI. De lezer wordt inzicht gegeven in de complexiteit van conventionele database-interfacing. In hoofdstuk 10 volgt een case study van de CFI-DRPI interface met gebruik van OPI. De eerste metingen geven aan dat de prestaties van OPI voldoende zouden kunnen zijn om een alternatief te bieden voor de procedurele interface. In hoofdstuk 11 wordt het in de voorgaande hoofdstukken behandelde werk overdacht. Verschillende suggesties gebaseerd op de ervaring van de schrijver en een vooruitblik worden gegeven. Samenvattend is het de overtuiging van de schrijver dat de afhankelijkheid van een object-georiënteerde programeertaal acceptabel is omdat dit type taal ook effectief is bij implementatie van grote software systemen. De eerste resultaten geven aan dat als de VOIS een goed partitioneringsalgorithm gebruikt de prestatie acceptabel is. De VOIS vormt een goede basis voor uitbreidingen.
Acknowledgements

I wish to thank the following persons for contributing to this work:

My mentor, Prof. P. Dewilde for approving this research topic, for obtaining the necessary funding and for the contribution concerning the technical and administrative aspects of the research.

The members of the commission: Prof. M. Ciletti, Prof. W. Gerhardt, Dr. P. Groeneveld, Dr. A. Kunzmann, Prof. D. McConalogue and Prof. F. Rammig for reading and correcting the preliminary version of this thesis.

Dr. R. van Leuken for enabling the use of the hardware and software resources required to carry out the implementation of OPI, ANT and the CFI-interface, and for the technical and administrative support.

Dr. J. Annevelink for introducing me to object-oriented programming and the persistency problem.

Ir. K. Schot for the contribution to OPI, the collaboration in designing and developing ANT and the CFI-interface, and for the partnership in representing the TUD in JCF SP4.

Mr. P. Kist for the collaboration in implementing ANT and the CFI-interface, and for the hundreds of hours spent with me in the evenings and weekends planning and implementing OPI.

The following students: Mr. N. Simon for the design, and Mr. A. Sarotama for the implementation of the OPI concurrency mechanism; Mr. E. Marks for the design and implementation of the OPI partitioning manager; Mr. J. van Dijk, Mr. R. de Boer and Mr. E. Rieff for the implementation of ANT; Mr. E. de Boer, Mr. R. Dolsma, and Mr. A. Rijkelijkhuiizen for the implementation of the CFI-interfaces.

The computer system administrator, Ing. W. Tiwon for quickly solving various technical problems, also outside normal office hours.

Mr. Guillermo Cuppers for the assistance in formatting the final draft of this thesis.

Mrs. C. Boers, Mrs. C. Klein and Mrs. A. Russo for the secretarial support.
Biography

Matthew Neil Sim was born in London on June 12, 1955 and emigrated to Israel in 1963, where he finished his high-school education at Kibbutz Eilon in 1973, specialized in agricultural machine mechanics. During 1973-1976 the author did obligatory military service in the IDF as a para-medic.

Upon discharge, the author settled at Kibbutz Carmia and worked on the design, construction and maintenance of agricultural machines. In 1979 the author attended the Wingate Institute of Physical Education where he obtained the diploma of qualified swimming trainer and lifeguard in 1980.

In 1982 the author moved to Haifa city to begin studies at the Technion Institute of Technology, and supported himself by working in the emergency ward of the Rothschild hospital, and training a local swimming team. After obtaining the B.S. in E.E. in 1986, specialized in micro-electronics, the author spent the first half of 1987 traveling through Europe and finally settled in The Netherlands.

The author continued his engineering education at the Delft University of Technology (TUD) at the Network-theory section of E.E. and performed research for the E.C. commissioned ESPRIT 991 project on the subject of data storage. The author obtained the Dutch Ingenieur's degree in 1989.

Between 1989-1992 the author worked towards the Dutch Doctor's degree at the Network-theory section of E.E. under the supervision of Prof. P. Dewilde. During 1990-1992 the author also worked for the Delft Design and Test Center (DDTC) of the Delft Institute of Microelectronics and SubmicronTechnology (DIMES) under the supervision of Dr. R. van Leuken. During this period Mr. Sim participated in the JESSI Common Framework subprojects ESPRIT 7364 and 5082 and various CAD Framework Initiative (CFI) activities.

The author has been nominated to participate in the EDA Frameworks working group of TC 117, a technical committee of CENELEC, the European Committee for Electrotechnical Standardization, and has been invited to serve as reviewer for a technical journal.

Beside his software activities, the author participated in the design and implementation of self-timed handshake controllers applied to high speed applications [80].

The author is a co-founder of bitbybit, a new company specializing in multi-media applications, which intends to utilize the OPT system.