VERTICALLY PARTITIONED OBJECT-ORIENTED SOFTWARE DESIGN FOR DEPENDABILITY AND GOOD PERFORMANCE

Stephen Peter Hufnagel

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-87-02 January 1987

ACKNOWLEDGEMENTS

The Department of Computer Science, The University of Texas at Austin, is great. The faculty have been encouraging to and supportive of me with their personal time and departmental facilities. I would like to thank the members of my committee for their review of my research and their constructive comments. Dr. Browne showed me how to formulate problems and how to think abstractly and he inspired, encouraged, and guided me through my graduate school years. Dr. Lengauer encouraged me to write clearly and understandably. Dr. Martin had the patience to coauthor my first journal paper and was particularly helpful during the thesis review period. Dr. Novak provided me with insights into system design. Dr. Malek refined my understanding of the fault tolerance literature. Dr. Dale as department chairman and then as graduate advisor provided the administrative support to get me through the PhD program. Nancy Macmahon provided innumerable instances of help and encouragement at my most difficult times.

I would like to thank my supervisors and the administration at my place of employment, Applied Research Laboratories, The University of Texas at Austin, for their patience, encouragement, and understanding of me during my graduate years.

The task of formatting this document was greatly eased by the use of the Scribe document processor developed by Brian Reid and maintained by Unilogic, Ltd. The Scribe dissertation format definitions for The University of Texas at Austin were developed by Richard Cohen.

Stephen Peter Hufnagel

The University of Texas at Austin
May, 1987

ABSTRACT

The goal of this research was to improve performance within an object-oriented software system design. It was desired that the performance of object-oriented software be competitive with alternate software design techniques for designing dependable software systems.

This thesis presents a design approach that first partitions a software program by data structure to define object data types. Then for each object data type, an associated type manager is defined. The type manager contains the necessary software procedures to perform all required operations on the object's data structure. Type manager functions should exploit an object's individual semantic properties in order to provide application-specific fault localization, fault recovery, "operating system" routines, and "database" routines. Layered, system-wide operating and system-wide database systems are, consequently, no longer required.

Simplicity of recovery results from the requirement to commit to the object's state at specific safe recovery points, defined by the completion of a type manager function. The type-specific functions and simplicity of recovery help to provide an overall performance improvement.

A software program is composed of a set of object type managers running on an abstract machine. The type managers may create and/or maintain instances of data objects. The abstract machine provides scheduling of and access to the computer's resources and execution time binding among object type managers.

This thesis presents quantitative simulation results to prove that the use of the proposed design approach can result in software that is an improvement over previous attempts to balance dependability and performance while maintaining high software comprehensibility.

TABLE OF CONTENTS

Acknowledgements iv

Abstract v

Table of Contents vi

Chapter 1. INTRODUCTION 1

1.1. Research Goal 1
1.2. Problem Statement 2
1.3. Scope 2
1.4. Thesis Plan 3
1.5. Design Approach 4
1.6. Design Overview 5
1.7. Design Analysis 6
1.8. Conclusion 8

Chapter 2. BACKGROUND 10

2.1. Definition of Terms 10
2.1.1. Function/Procedure 10
2.1.2. Module 10
2.1.3. Object 11
2.1.4. Object-Oriented 12
2.1.5. Type Manager 12
2.1.6. Vertically Partitioned 14
2.1.7. Process 15
2.1.8. Model of Execution 16
2.1.9. Model of Computation 16
2.1.10. Program 16
2.2. Fault Tolerance/Recovery Concepts 16

2.3. Transaction Concepts 21

Chapter 3. A DESIGN METHODOLOGY 26

3.1. Software Design Specifications 26
3.1.1. Overview 26
3.1.2. Amplification 28
3.1.3. Conclusion 32
3.2. Type Manager Requirements 32
3.2.1. Validity Testing 33
3.2.2. Atomicity of Operation 33
3.2.3. Distribution of Objects 34
3.2.4. Consistency Management 34
3.2.5. Synchronization 35
3.2.6. Integrity Management 36
3.2.7. Fault Localization 36
3.2.8. Fault Recovery 36
3.2.9. Access Control 37
3.2.10. Storage Management 37
3.2.11. Machine Interface 38
3.2.12. Summary 38
3.3. Type Manager Functions 39
3.3.1. Initialize 39
3.3.2. Terminate 39
3.3.3. Relocate 39
3.3.4. Recover 39
3.3.5. Create 40
3.3.6. Delete 40
3.3.7. Commit 40
3.3.8. Abort 41
3.3.9. Execute 41
3.4. Communication Among Type Managers 41
3.5. Design Aids 42
3.6. Design Steps 42

Chapter 4. AN APPLICATION ENVIRONMENT 44

4.1. Introduction 44
4.2. Application Requirements 44
4.3. Functional Software Specifications 47
4.3.1. Operating System 47
4.3.2. Database System 47
4.3.3. Application Software 48
4.4. Object Design Specifications 48
4.4.1. Abstract Machine 48
4.4.1.1. Bootstrap Function 49
4.4.1.2. TM Function 49
4.4.1.3. Storage Function 50
4.4.1.4. Resource Function 50
4.4.1.5. CPU ID Function 51
4.4.1.6. Communication Function 51
4.4.2. Type Managers 52
4.4.2.1. Type TM 52
4.4.2.2. Process TM 54
4.4.2.3. Program TM 55
4.4.2.4. Name TM 56
4.4.2.5. Switchboard TM 58
4.4.2.6. UNO Generation TM 59
4.4.2.7. Communications TM 59
4.4.2.8. Track TM 60
4.4.2.9. Display TM 60
4.4.2.10. Disk TM 61
4.4.2.11. DBM TM 61

Chapter 5. THE METHODOLOGY EVALUATION 62

5.1. Introduction 62
5.2. Goals 62
5.3. Scope 63
5.4. Approach 64
5.5. Software Metrics 65

5.6. Graph-Theoretical Description 66
5.7. Design and Implementation 69
5.7.1. Hardware Testbed 69
5.7.2. Workload 72
5.7.3. Software System Structures 72
5.8. Execution 73
5.9. Metric Data Description 73

Chapter 6. METHODOLOGY ANALYSIS 76

6.1. Quantitative Analysis 76
6.2. Qualitative Analysis 80
6.3. Conclusion 82

Appendix A. EXPANDED APPLICATION ENVIRONMENT 83

A.1. Introduction 83
A.1.1. Execution Environment 83
A.1.2. System Functions 84
A.1.3. Operational Environment 84
A.1.3.1. Physical Environment 85
A.2. System Requirements 86
A.2.1. Distributed System Subfunctions 87
A.2.1.1. Interprocess Communications 87
A.2.1.2. Resource Management 87
A.2.1.3. Security 88
A.2.1.4. Configuration Management 88
A.2.1.5. Database Management 89
A.3. Assumptions 90

Appendix B. METRIC LITERATURE REVIEW 91

B.1. Introduction 91
B.2. Software Metrics 91
B.2.1. Complexity Measure 95

Appendix C. EXPERIMENTAL RESULTS 98

C.1. CASE 1: Cold Startup 99
C.2. CASE 2: External Messages Received 104
C.3. CASE 3: External Messages Received 108
C.4. CASE 4: CPU 1 Failure 112
C.5. CASE 5: External Messages Received 116
C.6. CASE 6: CPU 1 Recovery 119
C.7. CASE 7: CPU 2 Failure 124
C.8. CASE 8: External Messages Received 127
C.9. CASE 9: CPU 2 Recovery 129
C.10. CASE 10: CPU 3 Failure 133
C.11. CASE 11: External Messages Received 137
C.12. CASE 12: CPU 3 Recovery 139

Bibliography 143

Chapter 1

INTRODUCTION

1.1 Research Goal

The goal of this research effort was to specify and evaluate by simulation, some performance properties of a design approach for designing dependable software systems with high integrity while maintaining high software comprehensibility.

Computer system *dependability* is the quality of the delivered service such that reliance can justifiably be placed on the service [Laprie 85]; we include here that the software may be required to continue to function properly in the presence of hardware component failure(s). Dependability is frequently quantified by the measures of availability and reliability. *Availability* is a measure of the fraction of the time when a system is able to meet its functional specifications. *Reliability* is a measure of the continuous service accomplishment from a reference initial instant. *Integrity* is the degree to which a system meets its design specifications. *Performance* refers specifically to the response time for execution of system functions. Integrity and performance may be specified with respect to a spectrum of execution environments or to a specific execution environment.

1.2 Problem Statement

Layered general purpose software structures frequently result in poorly defined data structures. These poorly defined data structures can result in, (1) high software complexity and reduced software comprehensibility (2) interdependencies among data accesses that cause difficult and sometimes impossible recovery logic, and (3) a high performance overhead when providing for system recovery. Typically, layered general-purpose software structures do not take advantage of the available semantic properties of the particular application in which they are used. This often results in inefficient, complex, and cumbersome solutions to simple problems. Performance is classically improved by piecemeal introduction of specific case solutions in an *ad hoc* manner. The usual result is reduced software comprehensibility and increased software complexity.

1.3 Scope

This thesis will substantiate some of the properties of systems built according to a new design methodology. That is, the results presented here will establish that a system built according to the design methodology can have performance at least competitive with a system built according to the (conventional) layered functionally partitioned design approach. We will also demonstrate some of the capabilities of vertically partitioned object-oriented design for management of faults.

1.4 Thesis Plan

This thesis introduces a new concept called vertically partitioned object-oriented software system design that will be specified as a part of an overall software design methodology. Chapter 2 reviews the concepts drawn from the literature and used as a basis to create the new software design approach. Chapter 3 specifies the design methodology and justifies the design decisions that resulted in the specification. In chapters 2 and 3, we will also discuss the traditional software design problems of deadlock, cascading rollback (domino effect) during fault recovery, orphan processes (as a result of a system crash), and consistency of data within a distributed and/or multiprocessing environment.

A simulation has been conducted to demonstrate the efficacy of the methodology. Chapter 4 gives the requirements for a sample execution environment and a sample workload. Chapter 4 also specifies two models of execution and two models of computation. That is, two computer simulation programs were specified. The first program was specified using traditional layered functional software design techniques. The second software program was specified following the vertically partitioned object-oriented design methodology.

In chapter 5, the design, implementation, and execution of the simulator are described. The two computer programs were developed to gather metric data while each separately performed the computation graph traversal of the same application requirements following the two different software

design approaches. Appendices A and B contain a summary from the literature review that was conducted to determine the sample execution environment and metrics that were used within the simulation experiment.

Experiments were defined and executed to evaluate some properties of the two software systems. The basic framework of the experiments was to view the computations as a graph and to observe that computations can be formulated as the flow of control between the binding of actions to data objects. Chapter 6 discusses both quantitatively and qualitatively the properties of the vertically partitioned object-oriented design approach. The quantitative discussion is based upon the simulation results, presented in Appendix C. The qualitative discussion is based upon the inherent characteristics of the new design approach. The conclusions of this thesis are presented at the end of Chapter 6.

The remainder of this chapter will provide a summary of the vertically partitioned object-oriented design approach, a brief design overview and analysis, and some general conclusions. The concepts and claims presented in this chapter will be further developed and justified later in the thesis.

1.5 Design Approach

The approach will be to develop an object-oriented system design that will "vertically" partition the state of the system (i.e., by object type). *Vertical partitioning* means that each object type has all of the necessary logic to perform required operations on its data. The concept of *composite objects* (that is, objects hierarchically built from smaller objects) will be developed.

Historically, object-oriented systems have been built upon a functionally layered operating system with the result of poor performance in execution. Vertical partitioning of object-oriented systems is new and is expected to result in improved performance.

In this design approach, a system is first decomposed into a set of data object types, which have associated type managers. Each *type manager* contains the procedures that execute functions on the specific data values contained within its data object(s). The type managers may create and maintain instances of data objects. Users may augment the primitive set of type managers with application-specific type managers. A *program* is the set of system object types, user-generated object types, and object occurrences corresponding to these system and user-generated object types. The abstract machine provides scheduling of resources and dynamic run-time binding among type managers and between each type manager and its required physical resources of the machine.

1.6 Design Overview

In a typical layered hierarchical system, the hierarchical structure extends across the entire system, whereas the vertical partitioned object-oriented approach gives a separate hierarchical structure to each individual object type. Decomposition (or composition) of total processing is done on the basis of data type rather than on the basis of processing function. Data are stored, processed, and retrieved within objects by type managers. The application of functions on an object's data is atomic. *Atomicity* is the

requirement that an execution of a function on the data of an object must be completed or aborted. As a result, an object's state is only updated at the successful completion of the execution of a type manager's function on an object's data. The atomicity property provides safe fault recovery points at type manager domain boundaries.

Vertically partitioned composite objects provide a foundation on which one can build an associated fault model. The recovery model will typically contain fault detection, fault localization, and fault-recovery to provide for system reliability.

Protocols to provide for concurrency within and among composite objects and for recovery from failure within composite objects will be considered. An extended notion of a specification to include reliability, integrity, and performance will be discussed.

1.7 Design Analysis

The motivation for an object-oriented system design is to attain high integrity with increased comprehensibility and reduced software complexity. The classical reason that object-oriented designs are not used is that they are typically inefficient, due to a multi-layered structure built upon a layered operating system. Object-oriented systems traditionally result in a workload that contains a great deal of context switching and data movement, thereby reducing performance. This increased workload arises because of the smaller granularity of system structures resulting from object-oriented design. Much

data movement is among related data structures. Vertical partitioning lowers the flow of data across domain boundaries. (Intra-domain context switches are typically less expensive than inter-domain switches.) Also, traditional object-oriented designs have not taken full advantage of semantic information available about the object's functions and storage.

All mechanisms for monitoring integrity in the presence of faults are based upon redundancy. By vertical partitioning of composite objects, we allow the use of the semantic properties of object data by the type managers that act on the data.. Generally, semantic properties of data and data structures can be used to improve overall system performance. The object-oriented design often allows compile-time binding of functions to their associated data structures. Compile-time binding typically results in improved system efficiency. Decomposition of total processing on the basis of data structure cuts down the data flow and inter-domain context switches. Performance may also be improved due to the locality of functions and data within the object boundaries.

The use of the semantic properties of object data results in type-specific fault localization, fault recovery, "operating system" support routines, and "database" routines. Hence, system-wide operating systems and system-wide database systems are no longer required. Simplicity of recovery results from atomicity at the objects boundaries and from use of type specific fault detection and recovery mechanisms. Vertical partitioning limits the propagation of faults, localizes data, limits the interdependencies among objects for fault recovery, and enhances overall system security.

Many large system designs require distributed and/or parallel processing capabilities. An object-oriented design results in a software structure where advantage may be taken of the independence of system distribution and the degree of system multiprocessing. This property can be used to apply the resultant software design to a distributed and/or multiprocessing environment.

A commitment protocol is required at the completion of a type managers function to insure the atomicity property of type managers. The commitment protocol of hierarchically constructed composite objects provide a hierarchy of synchronization between concurrently running type managers. Composite objects inherently have composite data structures. As a result, inconsistencies will not arise among data structures. It will be shown that composite objects result in a structure in which (1) deadlocks are avoided, (2) a simple protocol to solve the orphan problem [Moss 81] is facilitated, and (3) fault recovery protocols can be easily expressed.

1.8 Conclusion

This proposed design approach provides a management of complexity by providing a modularity based upon data structure that has the property of reduced interdependency of software modules and data structures. We claim that, for an equivalent fault tolerance specification, this proposed methodology will result in a software package that is not only comprehensible but also reduces execution time due to overall implementation efficiency. The use of the methodology that is developed here is expected to result in software that is

an improvement over previous attempts to balance dependability and performance while maintaining high comprehensibility and low complexity.

Chapter 2

BACKGROUND

2.1 Definition of Terms

In this section a review and definition of the terms used in the methodology will be given.

2.1.1 Function/Procedure

A *function* is defined as the mapping of a set of data from one domain to another domain (i.e., functions are applied to data). A *procedure* is defined as the means by which a function is implemented (i.e., a procedure is performed or executed).

2.1.2 Module

The definition of module to be used is a case of the Parnas criterion [Parnas 72]. A *module* (e.g., the type manager of an object) performs all of the required actions on its data, and specifies the necessary pre- and post-conditions for acceptance of the results of those actions. The functions of a module are made available to other modules as procedure calls, and these procedure calls constitute the only access to the functions of the module. In particular, the data manipulated by the module is only made available to other modules by procedure invocations; other modules have no direct access to the location or the representation of any data used by the module. Modules

detect conditions which violate their specifications and prevent application of functions upon the module's data when the necessary conditions are not met. [Wulf 75]

A module when seen from a user's point of view is perceived as being a variable of some abstract data type. A module, when seen from the inside, is a set of (i.e., state) variables and a set of procedures [Cristian 82].

It is important that the source language version of the implementation reflect the module structure of the design. Note, however, that the interfaces among modules are often macros. The ultimate object code may be distributed quite differently than may be apparent in the source code. [Wulf 75] As an example, parameters or messages may equivalently be used to pass information between modules.

2.1.3 Object

An *object* is the analogue of a variable in programming languages; an object is the abstraction of a typed storage cell [Wulf 81]. It has a "value" or "state." Often the representation of an object will be constructed from a number of other objects; in this sense an object strongly resembles a "record" in a programming language. An object may be thought of as a triple <Unique-Name, Type, Representation>. The *unique-name* of an object distinguishes it from all other objects that ever existed in the past, exist in the present, or will exist in the future. The *type* of the object defines the type of the resource that the object represents (e.g., program, process, data) and the functions that may be applied by executing procedures associated with the

object. Similarly stated, a type is an abstraction of a class of objects; the abstraction specifies the operations that apply to the objects. [Cohen75]. The *representation* of the object is a data structure which has associated procedures that operate on the data.

2.1.4 Object-Oriented

An *object-oriented* software system is completely built of objects. Each object will satisfy the Parnas module criteria. Each object type has a "black box description". Each object's description gives the internal functional specification and external interface specification. Externally the representation of an object is not visible. A language must have four elements to support object-oriented programming: information hiding, data abstraction, dynamic binding, and inheritance [Pascoe 86].

2.1.5 Type Manager

Formally, a *type manager* defines the functions that may be applied to a class of objects of the same type. This definition is parallel to the programming languages definition of a manager [Kieburtz 83] and [Silberschatz 77] that was derived from a monitor [Brinch Hansen 73] and [Hoare 74] for the specific purpose of resource allocation. Informally, a type manager is a set of procedures that defines what can be done with the object's data (e.g., modify, read, write, delete). A type manager is an object that responds to procedure calls.

Type managers check formal parameter specifications (e.g., type specifications of parameters) against actual parameter specifications (e.g., those parameters that are passed between procedures of objects). Type

managers may have other type managers execute functions on their own respective data. Type managers can create and delete instances of objects. Type managers can have embedded processes. Each type manager is directly responsible for the mapping of the occurrences (i.e., copies of data structures and/or copies of itself) of the objects that it spawns to physical storage. Each type manager implements access control policies for the occurrences of its type and implements the necessary level of redundancy to ensure the level of fault tolerance given in its specification.

A type manager may support concurrent operations such that there may be multiple active processes manipulating the data managed by the type manager. When concurrent operations are allowed, consistency management of the data objects that the type manager controls is the responsibility of the individual type manager.

Tasks will be performed by issuing calls to an object's type manager with a request for a particular function to be applied. At each level of execution, functions of type managers are atomic.

System and application type managers are created, modified, and destroyed by a special system type-type manager. The object type manager hierarchy is shown in Figure 1.

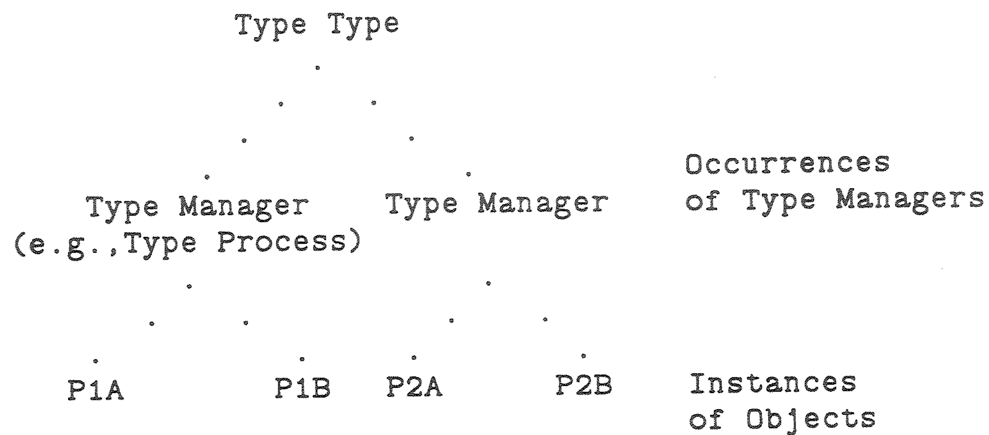

FIGURE 1: Object Type Manager Hierarchy

Occurrences of a particular type manager object type may be present at several different sites in a distributed system. It may be necessary for the type manager to coordinate the resource management among the occurrences of the type manager to avoid blockages and/or unnecessary delays.

2.1.6 Vertically Partitioned

A system is *vertically partitioned* if all functions applied to the object's data structure and all required support software are self-contained within the type manager of the object (i.e. an object's type manager does not share layers of support software with any other type manager.) That is, no component of one object will depend on the internal details of any other object. Only the object's type manager can change the state of its permanently stored data. Each type manager, if needed, performs all of the required functions of a traditional database management system and all of the required functions of a traditional operating system.

Each object is built upon the abstract base machine. The *abstract machine* provides scheduling of the resources of the machine and provides run time binding both among type managers and in addition between type manager and the machine's resources. Each type manager has direct access to the abstract machine.

The use of vertical partitioning allows a semantics which permits specific treatment of individual object types. It will be shown that a vertically partitioned object-oriented software program structure provides a natural mechanism by which we may compose elementary (primitive) operations on primitive data objects to form logical data structures and logical operations.

2.1.7 Process

Formally, a *process* is defined as an object for which there exists a type manager of type process. A process is a declared type that takes a program object, address space, and a processor to execute. A process is an object to which the function *execute* applies. A process is created (defined) by giving a type manager a program and an address space, and a processor.

Informally, a *process* may be defined as a schedulable stream of instructions with a program counter (PC) imbedded in some address space where names have been resolved. A process may invoke procedures defined by type managers. The process' locus of computation may flow from execution in one object's address space to execution in another object's address space.

Users may have many processes. By invoking the type manager of

type process, a process may create further processes which may be subordinate to the original process or may be of equal status.

2.1.8 Model of Execution

The *model of execution* contains an address space, a process type, an interpreter of the primitive units of operation, and the binding of the process to the interpreter.

2.1.9 Model of Computation

The *model of computation* contains primitive units of operation (e.g., abstract machine and a set of primitive type managers), schedulable units (processes), definition of address space, means of sequencing and means of synchronization, and interprocess communication facility.

2.1.10 Program

A *program* is an interpretable sequence of code. The model of computation must be satisfied for a program to execute within a machine that satisfies the model of execution. When programs are running, processes are executing functions on object state variables by invoking the object's type manager. Chapter 4 provides an example set of type manager specifications for a program.

2.2 Fault Tolerance/Recovery Concepts

This section will review the key definitions and concepts drawn from the fault recovery and fault tolerance literature.

A *system state* consists of records and devices with changeable

values. The system state includes system consistency constraints which are assertions about the values of records and about the allowed transformations of the values [Gray 81].

A *recovery point* is a point in time at which the state of a process is saved for possible regeneration [Wood 81]. A process expresses a *commitment* to a recovery point when recovery from that point in time is possible and the process no longer requires the data prior to that recovery point. The *recovery region* is the period of activity from the establishment of a recovery point to commitment to the next recovery point. Visually, *recovery lines* can be used to show the set of recovery points that a system will be rolled back to in the event that each of the processes initiates recovery actions [Wood 81]. A *latency interval* is the time elapsed between a manifestation of a design fault and a detection of the consequences of this manifestation [Cristian 82].

A *direct propagator* relationship takes place when the occurrence of a fault in one recovery region requires the rollback to a recovery point not in that region [Wood 81]. An *indirect propagator* is a direct propagator or a propagator that recursively leads to a direct propagator relationship. A recovery point RP is a *potential recovery initiator* of a recovery point RP' *iff* the RP is active and is an indirect propagator to RP' [Wood 81]. A *safe recovery point* is one which will not generate further recovery actions (domino effect) as a result of a recovery action initiated elsewhere in the system. The concept of recovery points, recovery regions, and recovery lines is a way of establishing commitment to the results of the execution to one or more transactions, as will be discussed later [Browne 83].

Fault recovery is initiated when an error is detected. The goal of the fault recovery mechanism should be to return the system to a consistent state from which processing may continue [Wood 81]. The recovered state should be a *safe state*. That is, there should be no recovery cycles which may lead to the domino effect. Additionally, the recovery mechanisms should not lead to deadlock. Finally, the recovery mechanism should be efficient. That is, it should not place such a large processing load on the system as to be impractical to use.

A *forward recovery* is the recovery sequence in which the final state of the recovery set is different from the initial state. Forward recovery is based upon knowledge of the semantics of the procedure that is being executed and has to be explicitly programmed by the designer of the procedure. *Backward recovery* is the method by which the variables in the recovery set recover their prior state [Cristian 82].

Any fault recovery scheme must deal with one or all of the following: 1) the standard execution domain, 2) the anticipated exception execution domain, and 3) the unanticipated exception execution domain [Cristian 82]. The *standard program execution domain* is executed if a program receives correct input data and each module executes properly to a normal completion. When a fault occurs, then the program will execute in an exception domain. The fault may be due to out-of-bounds input data, improper software design, or faulty hardware. The exception domain may be partitioned into the anticipated exception domain and the unanticipated exception domain. In the

anticipated exception domain faults can be predicted and, conversely, in the *unanticipated recovery domain* faults can not be predicted.

Two mechanisms may be used to implement fault recovery: (1) programmed recovery and (2) default recovery. *Programmed recovery* has specific user generated code placed within the module to deal with one or more faults. *Default recovery* has a general system mechanism used to back up to a recovery point, replace the module state information with consistent data, and reprocess. Programmed exception handling may generally be used for faults within the anticipated exception domain and some form of default recovery must generally be used to recover from faults in the unanticipated exception domain. In order to obtain highly reliable software both programmed exception handling and default recovery should be used in combination [Cristian 82].

Within a module, two fault recovery options are available: (1) masking and (2) signaling [Cristian 82]. *Masking* detects and corrects a fault within the module. The fault is not made visible to the calling module except possibly as an indication that a fault was located and corrected for error logging purposes. If a procedure R can provide its standard service, in spite of a lower level exception E which is propagated in R, we say that the propagation of E is masked by R. *Signaling* is the condition in which a fault is detected but recovery is not provided within the module boundary. The problem is passed to the calling module for possible resolution. Whenever possible, the responsibility of coping with faults specific to each interpretation level of abstraction need to be taken at the level where the fault occurs.

A recovery protocol must ensure that the system reverts to a consistent state in the event that one or more processes initiate recovery action, and it must support the determination of recovery point safety. A successful recovery protocol should maintain a record of the direct propagator relationship between recovery points and invoke recovery to all recovery points which are linked by the direct propagator relationship. When recovery is invoked by one process, the protocol should prevent cycles of dependencies (domino effect) by avoiding potential recovery initiator relationships. The protocol should avoid deadlock. Finally a recovery protocol should minimize the message passing and storage overheads associated with the protocol, and strive for conceptual simplicity.

Recovery from faults in the presence of concurrent processing is a difficult problem. The difficulty arises from dependencies between the concurrently executing processes which may cause cascading rollback upon the occurrence of faults. This problem occurs when recovery is based upon discrete state recovery points and logs of processing [Browne 83]. [Reed 83] gives a continuous state history protocol, where each binding of an action with a data object and the changes effected are recorded. In Reed's protocol cascading of rollback cannot occur under the assumption that a fault is immediately detected.

In their papers, [Wood 81] and [Russel 80] have given detailed analyses of mechanisms for determining valid recovery points for a dynamically varying set of interacting processes. The mechanisms involve

elaborate and complex procedures for keeping track of which recovery point is the currently valid recovery point for the currently executing set of processes. The recovery manager must, in essence, maintain a dependency graph for the interacting processes at run time. A fault occurring in a properly formulated recovery region within a process cannot invalidate current or past processing within another process.

In summary, any effective recovery protocol should (1) provide state consistency, (2) provide recovery safety, (3) avoid deadlock, and (4) be efficient [Wood 81]. In the next section, it will be shown that a transaction based software system makes an incremental step towards meeting these criteria.

2.3 Transaction Concepts

This section will briefly review the key definitions and concepts drawn from the transaction literature.

According to Gray [Gray 81], a *transaction* is a composition of primitive system actions which has the properties of atomicity, consistency, and durability. The *atomicity* property requires that the composite action taken is either completed or not completed. The effects of the composite action are not visible externally to the scope of the transaction. The *consistency* property requires that the actions of the transactions maintain satisfaction of the constraints defined for the data. The *durability* property requires that the effects of a completed transaction will not be undone. (The effects can only be changed by another transaction.)

A transaction may be viewed as a transformation of a system state. Each transaction is defined to have exactly one of two outcomes: committed or aborted [Gray 81]. Once a transaction is *committed*, its effects can only be altered by running another transaction. An *aborted* transaction has no effect on the system state. The *commitment interval* is the time interval between the beginning and the end of a transaction [Cristian 82]. It should be noted that a transaction binds a sequence of elementary actions to data items at execution time to create an indivisible action on a logical data object [Browne 83].

Actions on entities (system states) are categorized as [Gray 81] unprotected, protected, or real. An *unprotected* action need not be undone or redone if the transaction must be aborted or if the entity value needs to be reconstructed (e.g., a scratch disk). A *protected* action can and must be undone or redone if the transaction must be aborted or if the entity value needs to be reconstructed (e.g., a DBMS system). A *real* action cannot be redone (e.g., a cash dispenser).

From the beginning to the end of a transaction, real or unprotected updates of the state must be kept hidden from other transactions until the transaction is committed. One must stabilize the records which a transaction reads and keep them constant until the transaction commits (e.g., rereading the same record must give the same result). The method of synchronization generally used between transactions is two-phased locking with end-of-transaction commit [Gray 81]. This method results in a binding of a sequence

of actions to data by establishing appropriate locks on the items in the transaction and holding these locks until all actions have been completed. The two-phase method of synchronization can have a substantial overhead by reducing the degree of possible multiprocessing when the lock granularity is large.

Simple single-level transactions have primarily been used for simple short action sequences. In a single-level transaction, any failure of a sub-unit of processing will cause the entire transaction to abort. Consequently, simple single-level transactions may not be an adequately powerful tool for the formulation of complex processing structures over extended periods of time. When processing spans over several concurrent streams of execution then component transactions within the instruction stream may have to be rolled back for recovery of faults. These circumstances have led to the concept of nested transactions [Moss 82].

Nested transactions imbed transactions within transactions by applying transaction synchronization and recovery concepts within transactions as well as among them. Nested transactions allow the establishment of smaller indivisible actions within the scope of a larger externally visible durable action [Browne 83]. If one of the enclosed sub-transactions fails then the enclosing higher level transaction can recognize this as an exception and take appropriate action such as initiation of an alternative computation of the same action [Browne 83]. The nested transaction structure may be particularly useful when the computations involve algorithms which are robust in their application [Browne 83].

Shadow version recovery occurs when a transaction begins to hold a write lock for data and a backup copy of data is made. When a transaction commits, its parents (if any) may inherit some of the committing child's data versions. A top-level transaction that is ready to commit will know the identities of all nodes it affected through its descendants.

Children and parent transactions are connected when parents inherit locks as children complete. In a nested transaction system, if a child fails (aborts), the parent is not required to abort. This permits parents to attempt their own recovery, by retrying the action or using some other method. Locking rules in nested transaction systems provide for inheritance of locks [Moss 81]. Nested transactions permit safe concurrency within as well as among transactions, and enable transactions to fail partially in a graceful and controlled manner.

In a nested transaction system, inconsistencies may occur within the set of nested transactions. One way to detect these inconsistencies is to pass to the ancestor node a list of the descendant transactions committed and pending commitment. The superior node checks the list against a reference list. If the lists differ, a crash must have occurred at some earlier time. Transactions thought to have committed may be lost or transactions may be left as orphan transactions, in an indeterminate state. In this case everything must be undone and the top-level transaction must abort.

The advantage of transactions is that consistency relationships are

usually defined over actions and data object definitions which are more complex than the usual system primitives of reads and writes of single data items. This advantage is important because systems must maintain consistency both in normal concurrent processing and in the presence of faults.

Default exception handling based on automatic backward recovery can provide effective fault tolerance for design faults with latency intervals contained within the commitment intervals associated with transaction executions. This exception handling cannot cope with design faults of latency intervals which stretch over successive transaction executions [Cristian 82].

In summary, nested transactions form a basis for building distributed systems because they encapsulate the synchronization and failure properties of distributed systems in a very clean and usable way [Moss 81]. Unresolved problems in a transaction-based system are (1) how to provide for transactions over a long period of time when the latency interval is greater than the commitment interval, (2) dealing with orphans, (3) synchronization between transactions, (4) the inefficiency of run time binding.

It will be shown that the approach presented in the following chapter is an improvement over the achievements of the transaction approach discussed in this section.

Chapter 3

A DESIGN METHODOLOGY

The first section of this chapter will consolidate the key concepts brought forth from the literature into a vertically partitioned object-oriented design approach as a set of software design specifications. Justification for the specifications and design options will be discussed. The second and third sections will discuss the requirements for and functions of a generic type manager. The fourth section will discuss communication among type managers. The fifth and sixth sections will outline a set of design aids and design steps, respectively, that are generally useful in developing this type of software. Taken as a whole, these sections define a methodology for designing vertically partitioned object-oriented software programs.

3.1 Software Design Specifications

3.1.1 Overview

The approach is to develop an object-oriented system design that "vertically partitions" the state of the system by type. The concept of composite objects is included. Decomposition of total processing is done on the basis of data structure rather than on software functions. Data is accessed and manipulated within objects by type managers. Atomicity must be maintained by the type manager's procedures.

Atomicity implies that the invoking of an object's type manager will have one of two outcomes: a new state will be created and be committed to or there will be no change to the object's state. Each externally accessible function or procedure of the type manager will return, as a minimum, an exception flag, that will indicate a completed or aborted execution. A temporary state of pending commitment is possible, while waiting for higher level objects to commit. Once an object's state is committed to, its effects can only be altered by invoking a compensating action. During a temporary state of pending either the object's original state (i.e., possibly with a time tag) may be used for additional accesses, or access to the object's state information may be blocked, awaiting either the indication to commit to a new state or the indication to return to an earlier state.

Execution of a type manager's procedure may involve calls to procedures of other type managers. Each type manager must have the property of implementing indivisible actions. Type managers must recognize the failure of a given call from the exception flag that is received and use alternative computational procedures that may be provided by the called type manager [Browne 83].

The hierarchical definition of composite objects in terms of other objects must in each case guarantee the atomicity, consistency, and durability of the objects. Each type manager must implement correct concurrent access to the data of the object controlled by its procedures to maintain data consistency both during normal execution and when recovering from faults.

The type manager for each object may provide an undo and a redo procedure to allow fault recovery. Undo and redo must be restartable; that is, if the operation is already undone or redone, the operation should not change the object's state. Because the semantics of the application are known, undo operations can be specific to the fault.

Functions manipulating an object should exploit the object's individual semantic properties in order to provide application-specific fault localization, fault recovery, and "operating system" and "database routines". This is the strength of the approach of vertical partitioning. Software reliability, integrity, and performance specifications will dictate the extent of each object's fault detection, localization, and recovery logic.

A system is composed of a set of type manager objects installed on an abstract machine. The type manager objects may create and maintain instances of data objects. Each type manager contains all of the procedures that are needed to execute functions on the specific data values contained within its data object(s).

3.1.2 Amplification

A hierarchical object-oriented program structure is used because it provides a simple means by which programs may be structured. People are used to thinking about object abstractions. As in everyday life, physical objects are made up of smaller objects and as such the methodology includes component objects. The hierarchical use of composite objects in terms of other objects is analogous to the concept of nested transactions in database

management systems, because in each case the properties of atomicity, consistency, and durability must be guaranteed. These properties are the basis upon which a simplified fault recovery model may be built. Composite objects provide a variable granularity of data that provides design control over data consistency, software integrity, and fault recovery.

The fundamental requirement for the implementation of high integrity programs is the ability to construct atomic operations of arbitrary size on data objects of arbitrarily complex structure. Definition of objects through definition of their type managers amounts to early binding (at system definition time) of a sequence of elementary actions to elementary data objects to create indivisible actions on logical data objects [Browne 83]. The type manager of an object must contain the necessary mechanisms for consistency management and fault-tolerant scheduling.

The use of data abstraction in program development leads to programs which are structured into a hierarchy of modules. Visually, such a hierarchy may be represented by an acyclic graph. An acyclic recovery graph may be used to show the avoidance of the domino effect in recovery.

Simplicity of recovery results because the vertical partitioning of objects minimizes inter-object dependencies. Analogously to transaction systems, the invocation of the type manager of an object causes a state transformation which has the property of atomicity (all or nothing), durability (results survive failures), and consistency (a correct transformation, e.g., the procedure must obey legal protocols).

Macro software objects built from a subset of possibly reusable objects may be provided. A macro capability facilitates common software development. When the performance and integrity specifications dictate, the common software base can be optimized for each object's application.

In building a fault detection and recovery model upon a vertically partitioned, hierarchically structured, object-oriented structure, the key concepts brought forth from the transaction systems approach are atomicity and nested transactions. Analogies between the composite object-oriented structure described here and nested transactions as proposed by Moss [Moss 81] may be made, such that many of the recovery techniques developed for nested transactions are directly applicable to an efficient fault recovery mechanism here. Additionally, the early binding and semantic knowledge available from the hierarchical object structure will provide an execution time advantage in efficiency due to prior knowledge [Garcia 83].

Objects are hierarchically structured. Commitment of higher-level (ancestor) objects must wait until all lower-level (descendant) object calls have completed and tentatively committed. Should the higher-level object decide to abort execution, all lower level objects must also abort execution and return to a previous safe state. The type managers of the various objects in the system must contain the recovery and/or control procedures necessary to allow overall safe state recovery. Additionally, the type manager will implement correct concurrent access to a database of the objects defined by its functions and procedures.

For high reliability and high integrity applications, each object's type manager must maintain consistency at the object level both in normal concurrent processing and in the presence of faults in the underlying execution machine. Each type manager must deal with system failures and media faults, as appropriate. An independent decision can be made for each type manager as to how to implement recovery, based upon the circumstances of the individual type manager. This approach should lead to more efficient execution and performance since the context of the operation may be considered in making design decisions.

As an example, within an object either logging or recovery blocks may be used to allow fault recovery. *Logging* keeps the current state and an old reference state of each object, and a history file of differential state changes, called a log. Logging allows the object's type manager to reconstruct the object state from the old state plus the log. For *recovery blocks*, when the object is invoked, a copy of an entire relevant state history of the object is made. The recovery block allows the object type manager to return the object state to the old state. The decision of whether to use logging or recovery blocks is based upon implementation efficiency for specific systems or applications.

3.1.3 Conclusion

The functions and procedures of the object type managers can be viewed as implementing a set of transactions [Browne 83]. Analogously, each hierarchical level of abstract objects can be viewed as implementing a set of nested transactions. The basic difference between the transaction and object view is the time at which the sequence of elementary operations is bound to the set of data objects. The object view binds a sequence of operations to a set of data items at compile time while the transaction view binds the operational sequence at run time [Browne 83].

Whenever the need for indivisible actions on logical data objects can be foreseen and early binding can be provided, an increase in efficiency will be possible. Specificity in the choice of localized recovery mechanisms allows specific fault recovery that aids in overall system efficiency. For example, whenever the semantics of logical operations can be used to allow programmed exception handling, early binding can allow a simplified forward recovery from faults. Finally, partitioning at object boundaries limits the effect (e.g., access range) of functions, reduces the propagation of faults, and enhances overall system security.

3.2 Type Manager Requirements

In this section we define the requirements that should be considered for each type manager, as is appropriate to the specific application.

3.2.1 Validity Testing

Validity testing of parameters is the responsibility of the individual type managers. In addition to type testing, boundary condition testing of input parameters is possible. This represents a form of early fault localization that may help locate problems before they propagate among some or all of the type managers.

3.2.2 Atomicity of Operation

Objects may be composite. Within hierarchically constructed composite objects, atomicity must be maintained at each object level and at the highest level.

Each object's type manager procedure must return a status flag. A status indication of "aborted" will normally be accompanied by an associated exception flag that indicates the reason for the aborted action.

The object's commit control flow can result in data inconsistency and orphan processes [Moss 81], in the eventuality of a processor crash during this sequence. The window of vulnerability to crashes during the commit sequence between objects should be minimized. The protocol semantics are a design issue that impacts performance and software integrity.

3.2.3 Distribution of Objects

It is the responsibility of the type managers to provide for the distribution of objects over processors. Some or all of the objects may be centralized or distributed. As an example, one design option provides for highly reliable distributed objects with redundant copies of all data. This typically results in a high communication overhead and complex inter-processor protocols. Alternately, objects may be centralized. This design results in a potential single point of failure. The type managers may distribute the operations in accordance with the data consistency, program integrity, and overall performance requirements.

3.2.4 Consistency Management

Consistency management is the direct responsibility of each type manager. A uniform approach to consistency management is not required. Depending on the specific type manager application, an appropriate mechanism may be used. As an example, infrequently used type managers may disallow concurrency and use a simple lockout protocol while a frequently used type manager may support concurrency and use a version of a continuous state history protocol as described by [Reed 83].

There are various commit protocols that might be considered such as the two-phase locking with end-of-operation commit protocol. In this protocol a prepare to commit is given, an acknowledgement is received, a commit is given, and a final acknowledgement is received. Alternately, one of the one-phase commit protocols might be considered. In one-phase locking, a commit is given and an acknowledgement is received. A variation called one-phase

locking with presumed commit gives a commit and the positive commit is presumed. A reply is issued, within a prescribed time, only when an execution abort occurs and commit is not possible. Alternately, one phase locking with presumed abort gives a commit and an execution abort is presumed, unless an explicit commit is received within a prescribed time.

The simple one and two phase protocols only require data replication of the initial state data while the continuous history protocol requires a version (environment number) or timestamp associated with the data objects and multi-versions of replicated data. In a multi-version system, a scheme is necessary for purging unnecessary old copies of replicated data, as described by [Reed 83].

There is a large literature on consistency management protocols [Bernstein 81]. Within each type manager different protocols may be used in accordance with the semantics and requirements of that object's application.

3.2.5 Synchronization

If an object's type manager supports concurrent execution, the object's type manager must provide for synchronization within the object. The two-phase locking protocol is generally desirable because it allows each participant to unilaterally abort prior to the commit.

Hierarchically composite objects inherently provide locking of data at multiple levels of granularity. This is true because objects do not share data. An object is either solely responsible for its own data or it requests data from another object's type manager.

Operations that manipulate the data of an object are provided by the type manager of the object. Localized operations within objects should minimize the exchange of data between objects. Communications between objects should primarily contain commands and names. This is a very good security feature of well designed object-oriented systems.

3.2.6 Integrity Management

Integrity management of an object's data is provided at the object boundaries in that one object's type manager cannot directly access the internal structures of objects of another type. Integrity management within the object boundaries is the responsibility of the type manager. The degree of integrity management within each object's boundary is a design issue, depending upon the system integrity specification.

3.2.7 Fault Localization

Fault localization should be done at the appropriate composite level of the hierarchy of objects. The degree of fault localization should conform to the degree of abstraction being handled at each level.

3.2.8 Fault Recovery

Fault recovery can be done at the object level at which the fault is located, or the fault indication can be passed up to a higher level. Fault recovery in the anticipated exception domain may provide for forward recovery. Faults in the unanticipated exception domain can be handled by default recovery that resorts to some form of abort (backup recovery) and reprocess.

Faults may be either corrected and masked at the level where they occur or propagated to a higher level for resolution within the composite object structure. If concurrency is to be supported, an integrated unanticipated (i.e., subsystem crash) fault recovery backup capability and version or timestamp mechanism for concurrency control may be desirable. In this case a continuous state history protocol as described by [Reed 83] may be needed. The protocol described by [Reed 83] is attractive because it integrates the requirements of concurrency control and fault recovery into a single mechanism that is directly controlled by the individual type managers.

Anticipated fault recovery is best handled by the procedures that operate on the data of objects; unanticipated fault recovery is best handled by a separate type manager procedure. Crash recovery is the specific responsibility of the individual type managers.

3.2.9 Access Control

Access control can be implemented at the object level to provide maximum security, or some more lenient approach to access control can be implemented to improve system performance. Various schemes such as capabilities or tokens can be implemented.

3.2.10 Storage Management

The vertical structuring of the object system dictates that all storage management for the respective data structures of the object is the responsibility of each type manager. One type manager may not directly access the data of any other type manager. This means that the traditional data access features of an operating system and of a data base system are

provided at the type manager level rather than at the system level. Reusable code may be developed and used for many applications while specifically tailored code may be preferable in certain high-use objects. The requirements of the specific application should dictate whether simple or complex storage management techniques are provided.

3.2.11 Machine Interface

The specific machine implementation will provide an abstract machine. The abstract machine provides scheduling of machine resources and binding both among type managers and between type managers and the machine's resources. Each type manager will provide its own interface to the abstract machine.

3.2.12 Summary

The key concept in a vertically partitioned object-oriented design is that each object type is considered a self-contained unit that is accessed by the procedures of its type manager. For a particular application, each of the type manager procedures may be tailored to suit the specific need of the application. Early compile time binding of functions to data is provided at the object level. Late run-time binding may be necessary among objects because of the possibility of relocation in a dynamic distributed environment.

3.3 Type Manager Functions

In this section a set of commonly required type manager functions will be described to provide a template for the design of objects and their respective type managers. Depending upon concurrency and recovery requirements, a version and sequence number or timestamp may be desirable for all type manager procedure entry points.

3.3.1 Initialize

The initialize function must preset or set up data structures before the first execution of any functions on the data structures.

3.3.2 Terminate

The converse of initialize, the terminate function provides for the orderly shutdown of an instance of a type manager that is no longer desired.

3.3.3 Relocate

The relocate function provides for the orderly moving of one or more instances of a type manager's data from one system to another for distributed systems.

3.3.4 Recover

The recover function provides the recovery logic to recover from expected or unexpected failure during the execution of a function of the type manager. There may also need to be one or more recover functions for dynamic reconfiguration in a distributed execution environment in response to hardware status change (i.e., hardware failure or hardware recovery).

3.3.5 Create

The function of create is to install an instance of a requested object. This is the mechanism in which composite (descendant) objects are created. The objects may be static and long lived or dynamically changing depending upon the operational environment. All binding among objects is done by installing each type manager object in the abstract machine. The function of create includes the request of storage from the abstract machine.

3.3.6 Delete

The function of delete is to remove an instance of the requested object. This is the mechanism in which composite (descendant) objects are eliminated. The delete function must update the abstract machine and return storage to the abstract machine.

3.3.7 Commit

Within a composite object structure, the function of commit is to commit a version of the object and all of its descendant objects to the pending object's state. This entry, in conjunction with abort, is the mechanism that provides for atomicity. There may be a separate commit entry corresponding to each of the other type manager entries (excluding abort).

An object version can be discarded as a part of the commit protocol. If an object commits to a particular version in a sequence of composite object actions and one or more of them aborts, then the object version may also have to be discarded. Various commit protocols are available depending upon the consistency guarantee that is desired. As an example, a two phase commit protocol with presumed abort is very safe, but may have a high performance overhead for frequently used type managers.

3.3.8 Abort

The function of abort is to retract a ready to commit object version. This function represents the alternate to commit, while guaranteeing atomicity. If an irrecoverable fault or an orphan process is located, it may be necessary to abort an operation and return to a previous consistent state.

3.3.9 Execute

The function of execute is to execute some type specific function(s). This entry or set of entries provide for the type specific functions of the type manager. In some applications, that transform object data, an undo function may be desirable as a separate entry, due to the atomicity of operations.

3.4 Communication Among Type Managers

Communications among type managers and/or processes is an important function. It can be handled by either direct procedure calls or by processes that manage a mailbox system. A name manager and a switchboard (circuit) manager may be used in establishing communications. A name manager establishes a path to the present location of the object but does not guarantee that the object will remain at that location. Migratory objects may be tracked by either a forwarding address or a call to the name manager.

A switchboard manager establishes a two way communications path. A switchboard manager is analogous to a two way circuit. Both ends of the circuit must be updated or have the information available to update if there is a change in the connectivity.

3.5 Design Aids

The design steps and design tools sections will be brief with reference to the literature because these techniques are well documented. Data flow and control flow diagrams, data object tables, hierarchy charts, and computation structure graphs are graphical design tools that are generally useful to facilitate software design. Data and control flow diagrams are useful to visually analyze data and control flow as shown by [DeMarcos 78]. To determine natural object boundaries around data structures for a system design, a data object table based upon the modularization criteria of [Parnas 72] and further developed by [Duncan 84] may be used. Traditionally, hierarchy charts have been used to display program organization. A good reference for the use of hierarchy charts is [Buhr 84]. More recently, computational structure graphs are being used to detail out the characteristics, module interfaces, and control and data flow as described by [Browne 85].

3.6 Design Steps

A program can be best designed [Browne 84, Browne 85] by following these design steps.

- Map the software requirements statement into a partitioning of data objects. Determine the data objects and their interfaces with the objective of creating a modularization with maximum cohesion within the objects and minimum coupling among objects. Object structure diagrams are a useful design aid here.

- Define the type manager processes and the externally visible type manager functions controlled by each type manager. This step provides the classical black box description of each type, that is the internal functionality and external interfaces and parameters passed for the type manager of each object. Structure diagrams are

a useful design aid in doing this step. At this point the abstract machine interface is mapped into the overall design.

- Express the (trans)actions of the system in terms of the sequences of calls among type managers. This step shows the relationship among types for each externally visible system event by showing for each type manager process or type manager function, the sequence of external procedure calls. Control flow and data flow diagrams are useful design aids for performing this design step.

- Specify the protocols to be used among object procedures and processes. Control flow diagrams are a useful design aid in performing this design step.

- For each type manager map the type manager design to a specific modularization implementation. Hierarchical diagrams and a definition of the internal data structures are useful design aids at this step.

- For each type manager specify the detailed algorithms and processing within each object. Internal structure diagrams and/or a high order design language are useful design aids at this step in the design process. Additionally internal flow diagrams and internal control diagrams may be useful.

- Integrate the module design into a cohesive system design.

In summary, the design process should guide the user to look at data representation, to partition the program by data structures, and to define objects by the set of functions that act upon the data structures.

Chapter 4

AN APPLICATION ENVIRONMENT

4.1 Introduction

This chapter will define a set of requirements and two sets of associated specifications for a particular application. First the application requirements, which include an execution environment and workload description, will be given. Next the specifications for a traditional functionally layered design will be given, and then the specifications for a vertically partitioned object-oriented design will be given. These functional and object design specifications will be the basis for the evaluation experiments described in chapter 5.

4.2 Application Requirements

This section will describe the application requirements that will be used in making design decisions in the subsequent software specifications section. This section is a subset of the more complete requirements given in Appendix A. The abbreviations were made to simplify the experiment and to emphasize the more interesting aspects of the requirements with respect to the methodology evaluation. Much of this material is based upon [Tripathi 83].

The hardware configuration will consist of three computers connected

to two local area networks that are also attached to two disks, two operator displays, two communication (message) receivers and a system command unit. Communications between the hardware units will be by the passing of messages on one of the local area networks. The application programs will support external message reception of track-update messages by the communications units. One of the communications units will be enabled by the software program to format each track-update message for transmission to the communications processing function on one of the computers. There will be three software processing functions: (1) the communications processing function, which will receive track-update messages from a communications unit on the local area network, decode each message to create a track-report message, and send each track-report message to the track processing function. When the system is initialized the communications processing function will send a message to one of the communications units to make it the active unit. The other unit will stay in standby. (2) The track processing function will receive track-report messages from the communication processing function, will perform track update processing, will create display-update messages, and will pass the display-update message to the display processing function. (3) The display processing function will receive display-update messages from the track processing function, will create two display-command messages (400 words each), and will send the display-commands messages to both of the display units. When the system is initialized the display processing function will send a 4000 word initialization-commands message to each of the display units.

The system will provide for continued operation after and recovery from a single point of failure. That is, the system will automatically reconfigure itself to bypass a non-operational hardware unit and continue receiving and processing external track-update messages. If a non-operational hardware unit becomes operational the system will reconfigure itself to include the operational unit.

When all hardware units are operational the following system configuration will be used: 1) Communications unit 1 will be active and communications unit 2 will be in standby. 2) Local area network 1 will be exclusively used for communications among hardware units. 3) The communications processing function will be located in computer 1, the tracking function will be located in computer 2, and the display processing function will be located in computer 3. 5) Duplicate information will be placed on disk units 1 and 2. 6) Duplicate information will be displayed on display units 1 and 2.

If a computer becomes non-operational, the system will detect the failure and reconfigure itself to redistribute the processing function of the non-operational computer to the next higher numbered computer, modulo three. (i.e., if computer 1 fails then computer 2 will configure itself to do both the communications function and the tracking function.) If a computer becomes operational, after having been non-operational, the system will reconfigure itself to the fully operational configuration just described.

The following assumptions may be made: 1) Messages are reliably passed and messages will be passed in the order that they are sent. 2) The hardware units are "fail stop", that is, they will stop and provide an indication when a hardware failure occurs.

4.3 Functional Software Specifications

In this section the specifications for a layered functional software design will be given. These specifications will be brief because traditionally a layered functional design of software is used.

4.3.1 Operating System

Each computer will have a kernel operating system. Each operating system will be responsible for local configuration based upon the operability of the system hardware units. It will maintain a local and global status of the system configuration and exchange status between processors whenever a software reconfiguration occurs. The operating system will provide local area network access routines and pass messages among processes and the local area network. Interprocess communications will be done by message passing.

4.3.2 Database System

A database system will be provided to allow uniform access to disk data structures.

4.3.3 Application Software

The application software will be partitioned into three processes: a communications process, a track process, and a display process. The default system configuration is to have the communications process on CPU 1, the track process on CPU 2, and the display process on CPU 3.

4.4 Object Design Specifications

This section will detail the design specifications for a vertically partitioned object-oriented design to meet the requirements given earlier in this chapter.

4.4.1 Abstract Machine

We are generating a distributed system that integrates the normal operating system and database system functions. There is no specific underlying function of a distributed operating system in this system structure since the functions are subsumed by the selection of basic type managers. Any host machine that supports the set of type managers to be specified will have an abstract machine interface that provides binding among type managers and between type managers and resources.

The abstract machine has a scheduler that multiplexes the processor among the type managers and schedules the type manager's requests for memory and communications resources. The abstract machine will provide the processor number upon which the software is running. The abstract machine interface consists of requests for resources (i.e., CPU, local area network and memory), linkage between type managers, and requests for CPU identification

(ID). The abstract machine is also responsible for restarting the system and the startup of the type-type manager. Each abstract machine function will now be discussed.

4.4.1.1. Bootstrap Function

The bootstrap function is used to start or restart the system software. Actual recovery within the type managers will be performed by the individual type managers. This function bootstraps an object whose name is type and type is type that establishes the type manager operations.

Entry Name: `Bootup`

```
Parameters:
 O   Status indicator of results, if failure
```

Note: An I before a parameter descriptor indicates that the parameter is an input to the function and an O before a parameter descriptor indicates that the parameter is an output from the function.

4.4.1.2. TM Function

The type manager (TM) function of the kernel does the necessary linkage to install or remove type managers from the system so that the computer scheduler will cause the execution of the proper type manager procedures. The installation and removal of a type manager into the abstract machine is the responsibility of the type manager requesting the CPU resources.

Entry Name: `TM Installer`

```
Parameters:
 I  UID of the TM to be installed
```

```
 I   TM linkage block (i.e., entry points)
 O   Status indicator of results
```

Entry Name: `TM Remover`

```
 I   UID of TM to be removed
 O   Status indicator of results
```

4.4.1.3. Storage Function

The abstract machine's storage function allocates memory to type managers and keeps a directory of the ownership of these resources.

Entry Name: `Memory Request`

```
Parameters:
 I Caller UID
 I Amount of storage space (i.e., blocks)
 O Physical location of logical block(s) of storage
 O Status indicator of results of invocation
```

Entry Name: `Memory Return`

```
 I Caller UID
 I Amount of storage space (i.e., blocks)
 O Physical location of logical block(s) of storage
 O Status indicator of results of invocation
```

4.4.1.4. Resource Function

The abstract machine's resource function allocates disk blocks to be controlled by a type manager and keeps a directory of allocated blocks.

Entry Name: `Resource Request`

```
Parameters:
 I   Caller UID
 I   Number of blocks desired
 O   Starting block number
 O   Status flag - indicate the results of the invocation
```

Entry Name: `Resource Return`

```
Parameters:
 I   Caller UID
```

I Starting block number
I Number of blocks returned
O Status flag - indicate the results of the invocation

4.4.1.5. CPU ID Function

The abstract machine's CPU ID function uniquely identifies the computer number upon which the software is executed.

Entry Name: CPU ID

Parameters:
O Unique CPU ID

4.4.1.6. Communication Function

The communication function will provides for communication linkage between type manager and other type managers or the active local area network, so that type managers may pass messages.

Entry Name: Message Send

Parameters:
I Callee UID
O Linkage to destination
O Status

Entry Name: Message Receive
I Callee UID
O Linkage to source
O status

4.4.2 Type Managers

This subsection will specify the set of type managers that will be used to meet the application requirements given earlier in this chapter. The initialize, recover, and disable entries are common to all type managers and will not be described except when they have special properties.

4.4.2.1. Type TM

The type-type manager is the object in the system that manages "types" in the system, and it is the means by which new type managers are introduced into the system and unnecessary type managers are deleted from the system. The type-type manager is also responsible for configuration control for system initialization and system reconfiguration, in the eventuality of system failure or recovery.

The type-type manager can create active processes and it can create data objects with associated type managers to access or transform the data. The type-type manager is also responsible for the distribution of type managers.

Entry Name: Initialize

Function: Install the initial set of system type managers.

Parameters:
O Status indication of the results of the invocation

Entry Name: Install

Function: Install and initialize a type manager.

Parameters:

I UID of new TM
O Status indication of the results of the invocation

Entry Name: Remove

Function: Terminate and remove a type manager

Parameters:
I UID of the TM to be removed
O Status indication of the results of the invocation

Entry Name: Dead Remote CPU

Function: Provide local reconfiguration to compensate for the failure of a remote CPU.

Parameters:
I UID of failed CPU
O Status indication of the results of the invocation

Entry Name: Recovered Remote CPU

Function: Provide local reconfiguration to compensate for the recovery of a failed remote CPU.

Parameters:
I UID of failed CPU
O Status indication of the results of the invocation

Entry Name: Shutdown

Function: Shutdown the system

Parameters:
O Status indication of the results of the invocation

4.4.2.2. Process TM

Type process is a particular object type that controls execution of other type manager's functions. It takes schedulable tasks and executes the functions of control including scheduling and termination. By control we mean that the type manager manages the state of the process so that resources can be made available appropriately to those processes which can be executed. A type manager for type process takes an appropriately defined program object (see program manager), binds it to an address space, and executes operations (e.g., run, stop, checkpoint) on the program.

Entry Name: Process create

Function: Take a copy of program object, bind it to an address space, assign it a process UID, and begin execution.

Parameters:
O Process UID of process
O Status indication of the results of invocation

Entry Name: Process Kill

Function: Kill a process.

Parameters:
I Process UID
O Status indication of results of invocation

Entry Name: Process Pause

Function: Pause the execution of a process.

Parameters:
I UID of process to pause
O Status indication of results of invocation

Entry Name: Process Restart

Function: Restart a process that has been paused.

Parameters:
- I Process UID
- O Status indication of results of invocation

Entry Name: Process Checkpoint

Function: Establish a copy of the process's state.

Parameters:
- I Process UID
- O Checkpoint number
- O Status flags - Results of invocation

4.4.2.3. Program TM

The program type manager's function is to compile the program text and to link binary code into object structures. The bit stream of a type program object conforms to the syntax and semantics of the intended interpreter, which should be specified as part of its state.

Entry Name: Program Object Create

Function: Create a program object in compilable form.

Parameters:
- I UID of the program object source text
- I Object name
- I UID of object
- I Version number
- O State indicator (e.g., source only, compiled and ok, compiled but bad)
- O Status flags

Entry Name: Program Object Destroy

Function: Destroy a program object.

Parameters:
 I   Object UID
 I   Version
 O   Status indication of results of invocation

Entry Name: Program Object Compile

Function: Compile source code, create a listing of a
          source program, binary code,
          test for errors, and create linker table.

Parameters:
 I   Object UID
 O   Status flags

4.4.2.4. Name TM

The name manager is the type manager that creates unique names (UIDs) for all objects in the system and handles the symbolic name to UID mapping. The name manager must guarantee unique names across the application system.

Entry Name: UID Provide

Function: Create a unique UID.

Parameters:
 I Caller UID
 O New UID
 O Status

Entry Name: UID Name Associate

Function: Associate a symbolic name with an object's UID.

Parameters:
 I Object's UID
 I Symbolic name

O Status flags

Entry Name: UID Kill

Function: Remove an UID from the system.

Parameters:
- I UID
- O Status

Entry Name: UID Name Disassociate

Function: Disassociate a symbolic name from an object's UID.

Parameters:
- I UID
- I Symbolic name
- O Status

Entry Name: UID from name

Function: Given a symbolic name, return the object's UID.

Parameters:
- I Symbolic name
- O UID
- O Status word

Entry Name: Name from UID

Function: Return an associated symbolic name.

Parameters:
- I UID
- O Symbolic Name
- O Status flags

4.4.2.5. Switchboard TM

This type manager provides an "operation switch" and provides guaranteed paths to the type managers. The concept of a switchboard manager is that it is analogous to a two way circuit. Both ends of the circuit must be updated or have the information available to update if there is a change in the connectivity. The switchboard is called by the abstract machine (dispatcher) whenever the latter cannot satisfy a communications request from a type manager or from the local area network controller.

Entry Name: TM Locate

```
Function: Locate a type manager.

Parameters:
 I  Desired TM UID
 O  Linkage to type manager
 O  Status
```

Entry Name: Receive status

```
Function: Process a status-information message from a
remote switchboard type manager.

Parameters:
 I  linkage to message
 O  Status of invocation
```

Entry Name: Send status

```
Function: Process a status-request message from a
remote switchboard type manager.

Parameters:
 I  linkage to message
 O  Status of invocation
```

Entry Name: CPU down

Function: Process an advisory-message from a remote CPU.

Parameters:
 I Linkage to advisory message
 O Status of invocation

Entry Name: Reroute message

Function: Reroute a message.

Parameters:
 I linkage to message
 O Status of invocation

4.4.2.6. UNO Generation TM

This system wide type manager will provide the capability of generating unique numbers (UNOs) within a bounded range.

Entry Name: UNO

Parameters:
 O unique number
 O Status flags

4.4.2.7. Communications TM

This is the application type manager that accepts, decodes, and processes track-update messages from either of the communications units, creates track-report messages, and passes the track-report messages to the track type manager.

Entry Name: Initialize

Function: Send an initialization-command message
to external communications unit 1 establishing
the CPU location of the communications type manager.

Entry Name: Process

Function: Process a track-update message.

Parameters:
 I linkage to track-update message
 O status of invocation

4.4.2.8. Track TM

This is the application type manager that processes the track-reports messages, creates display-update messages, and passes the display-update messages to the display type manager.

Entry Name: Process

Function: Process a track-report message.

Parameters:
 I Linkage to message buffer
 O status of invocation

4.4.2.9. Display TM

This is the application type manager that processes track updates and services the operator display unit.

Entry Name: Initialize

Function: Send a 4000 word display-commands initialization message to the operational display units on the system.

Parameters:
 O status of invocation

Entry Name: Process

Function: Process a display-update message.

Parameters:
 I Linkage to display-update message

O status of invocation

4.4.2.10. Disk TM

Entry Name: Disk receive

Function: Process a message from a disk unit.

Parameters:
 I Linkage to message
 O status of invocation

Entry Name: Disk send

Function: Send data to a disk unit.
disk.

Parameters:
  I Linkage to disk-message

4.4.2.11. DBM TM

Entry Name: Get

Function: Get information from a data structure
for a type manager.

Parameters:
 I data name
 O data
 O status of invocation

Entry Name: Put

Parameters:
 I data name
 I data
 O status of invocation

Chapter 5

THE METHODOLOGY EVALUATION

5.1 Introduction

This chapter will first present the goals, scope, and approach of the methodology evaluation effort, and a brief review of the software metric study will be given. Then a description of the techniques of design, implementation and execution of the experiment will be reviewed. Next a graph-theoretical description of the simulation experiments will be presented. Finally a definition of the specific types of metric data which were collected during the experiments will follow. A case by case presentation of the experimental results will be presented in Appendix C. Chapter 6 will give the overall analysis of the methodology evaluation results.

5.2 Goals

The goals of the methodology evaluation effort were to quantify and to evaluate the performance-related properties of an object-oriented structured system and to compare these properties to similar properties of a functionally structured conventionally layered system.

5.3 Scope

The evaluation was confined to evaluating the performance impact which results from varying the software design approach. Two computer simulation programs were written in which the software design approach used to meet the same system requirements was the only variable which was changed. This resulted in a vertically partitioned object structured simulation program and a layered functionally structured simulation program.

System structure affects the performance cost of satisfying inter-data and inter-functional dependencies and controlling the execution of a computation. Past object-oriented system structures which have been embedded in conventional software environments have incurred increased execution control and recovery dependency satisfaction performance overhead costs which rendered their performance infeasible for actual use. A highly modular structure will be expected to incur higher costs for these functions because of a greater degree of partitioning of both state and functionality. The approach of vertical partitioning partitions both the data structures and functionality of the system so that the execution of a function on a given data object will have reduced "overhead" costs. The vertical partitioning also encourages the exploitation of an object's individual semantic properties to minimize the execution of both the functions on data and system "overhead" functions such as consistency management and recovery.

This study is confined, however, to comparison and evaluation of the costs of executing these "overhead functions" for both an object-oriented

system structure and a functionally layered system structure in an otherwise fixed execution environment. No advantage was taken of the possibilities of the vertically partitioned object-oriented system for using the properties of functions on objects to make the processing more efficient except for the case of fault localization and fault recovery, which is a function required by the application.

5.4 Approach

The evaluation of the methodology was done by simulation modeling. Two programs were written to evaluate execution behavior. The requirements and specifications for a sample application were given in chapter 4. The first program met the design specifications of the functional layered approach, given in chapter 4. The second program met the design specifications of the object-oriented approach, given in chapter 4. Both programs met the requirements for the application, given in chapter 4.

Both hardware and software were simulated. The programming language PASCAL was used. The simulation software was executed on a DEC 2060 computer. Transaction-based modeling was used. That is, sets of identical states (e.g., a CPU crashed, two CPUs were functional, etc.) were defined in the two simulator programs and identical transactions or sets of transactions were executed by the simulators. Each simulation program incorporated measurement data gathering for analysis of the designs.

The functionally designed model was developed first. When the

software program was completed and working to the author's satisfaction, the program was "frozen." A copy of the program was made and used as the basis to start the object design. The complete hardware simulation portions and metric collection portions of the original simulator were kept intact. The operating system code, database code, and application code were repartitioned and augmented to create object structures. Then the code was modified as needed to work within the object structures. Finally fault localization and fault recovery portions of the code were optimized as appropriate, to take advantage of the individual semantic properties of the object structures. No functional capability was introduced nor was any functional capability lost in creating the object designed program from the functionally designed program. Only the structures were changed.

5.5 Software Metrics

A software metric is a measure of the differences between the two design methodologies. As in software design, there should be a hierarchical structure in metrics, so that comparable properties may be compared at equivalent levels of abstraction.

The software metric literature was reviewed to determine if there were well-defined metrics for the evaluation of software systems which would lead directly to evaluation software performance properties. The results of this study are summarized in Appendix B. It was found that most software metrics relate to code structure, not to performance impact. A set of metrics were selected, which define the execution cost and thus the performance impact of the differences between the two system structures.

The metrics for evaluation of the "overhead" processing costs defined in section 5.3 are those which measure the amount of computational work done to compose "primitive" elements of computation into logical computation structures. These overhead costs result from the satisfaction of dependencies and from the control of flow. They also include system-related functions such as fault recovery. These processing overhead costs include the flow of data between units of computation and the costs for invocation of units of computation. The specific metrics which we chose include the number of CPU context switches, flow of data and control between software modules, number of changes of execution scope, and flow of data among CPUs and I/O units. These metrics will be described in section 5.9.

5.6 Graph-Theoretical Description

The program code for the simulation programs was extensive; hence, it was not presented in this thesis. Instead, a graph-theoretical description is given of the model of computation of the two simulation programs that were developed and used to evaluate the methodology of chapter 3.

Each simulation program may be represented by a computation graph; the *computation graph* is defined as an ordered pair $<V, E>$. V is a set whose members are called vertices and E is a binary relation on the set V. The members of set E are called (directed) edges. A *path* in the graph is a sequence of alternating vertices and edges where the set of edges are of the general form $e(n) = <a(n), a(n+1)>$ such that each a(i) is a vertex. In this application the *vertices* represent function execution domains (i.e., subroutines

or processes or hardware units that process data) and the *directed edges* represent potential steps in an execution path; that is, two vertices are connected by a directed edge *iff* the locus of computation can pass from the source vertex to the destination vertex.

For the experiment a computation graph is defined for the functional program and a separate computation graph is defined for the object program. A separate set of vertices and edges defines the behavior of each hardware unit. In each computation graph, the sets of vertices and edges representing the communications, system control, disk, and display hardware units are identical. The sets of vertices and edges representing computers 1, 2, & 3 differ and correspond to the functional and object systems software respectively. For evaluation of the methodologies, test cases were defined. Each *test case* specifies a set of system states and the control representation of a message type to be processed. For each test case, a binary state was defined for each hardware unit; that is, the unit was defined as being operational or non-operational. For each operational hardware unit, its associated vertex and edge set defined a part of the respective computation graph (i.e., resulting in one computation graph for the functional system and another computation graph for the object system). In this application, a path through a computation graph represents the execution flow defined by the processing of a particular transaction type by the system.

Each simulation program described in Section 5.6 defines a special computation graph called a metric graph; a *metric* is a function from some

abstract concept to some ordered set. For our purposes, a *metric graph* is a graph with a metric defined on the set of vertices and on the set of edges (i.e., a metric graph is a graph such that f:V --> N AND f:E --> NxN, where N is the set of natural numbers). Each vertex contains control logic and control information and/or data are associated with every edge. In our application, the metric value associated with each vertex is called a *context switch* and represented the performance cost of changing function execution domains. For each edge there are two associated metric values: *control flow* which represents the performance cost of passing control information between verticies and *data flow* which represents the performance cost of passing data between function execution domains (i.e., between vertices).

The experiments will first be described from the graph viewpoint. At each vertex in the functional dependency graph, the execution of the functions associated with the vertex and the edge that brought the execution flow to the vertex was performed and was followed by a transition to another vertex. For each test case and metric graph a functional dependency graph results; a *functional dependency graph* is a subgraph of the metric graph, defined by the resultant path in the metric graph, for a specific test case. For each hardware unit, the metric values were summed for the object system and the functional system resultant functional dependency graphs.

Now the experiment will be described from the program viewpoint. Each execution of one of the software programs can be represented by a functional dependency graph; the program executes the metric functions at

each vertex and makes the transition between function execution domains represented by the edges. When executed, each software module (vertex) first calls an accounting routine that updates the metric value sums (particular sums are associated with each hardware unit) of that vertex and the edge transition that caused that execution domain to be entered. The accounting routine also keeps a sequential history of the vertices visited; hence, defining the execution flow by a sequence of edge transitions.

5.7 Design and Implementation

Each software model simulated three components: a hardware configuration, the definition of a software system structure, and a workload. In addition each software model was equipped with an execution path trace capability and performance measurement software. The hardware structure, software instrumentation package, and the workload were identical for both the functional simulation and the object-oriented simulation. The only thing that was changed and evaluated was the two software system structures (vertically partitioned object-oriented structure versus functionally layered structure).

5.7.1 Hardware Testbed

The hardware testbed was designed to meet the requirements given in chapter 4. For each simulation program the same hardware testbed was simulated. In each case three CPUs were connected by two communications local area networks (LANs). Attached to the networks were two disks, two message communications (COMMS) units, two operator display consoles and a control console. Duplicate units of each type of failure critical node (i.e., CPU,

Comms, disk, network, display) were identical except for address. Identical units were specified to allow automatic software reconfiguration to provide for continued system operation in the event of the failure of a single hardware unit. Two interface design specifications (IDS) were written, one for each simulator. The hardware description and usage in each IDS was identical. The inter-process versus inter-type manager message formats differed between both simulator programs as a result of the different software structures. Figure 2 illustrates the hardware testbed design.

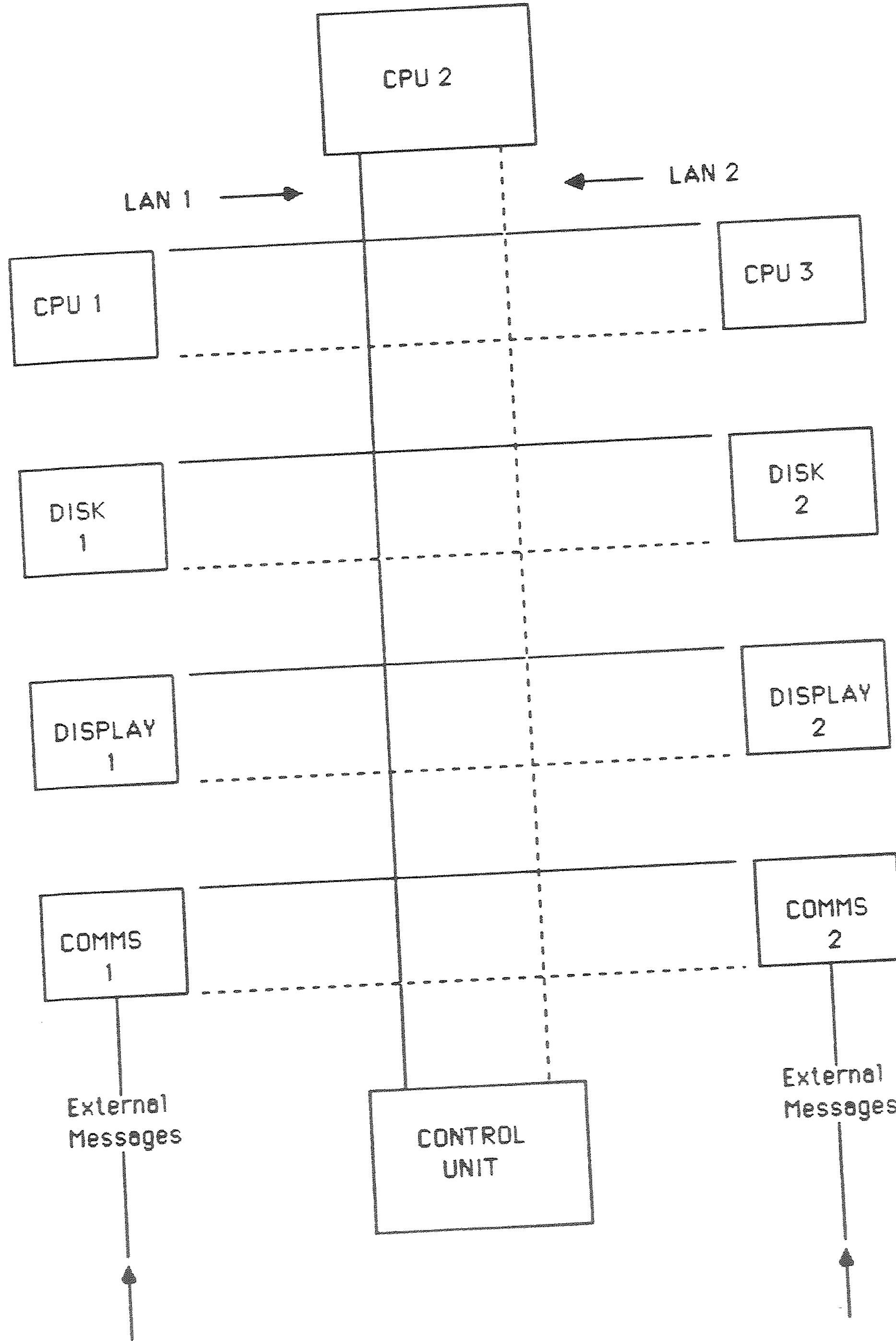

FIGURE 2 : Hardware Testbed

5.7.2 Workload

Twelve test cases were defined to evaluate the two software designs. A test case was composed of a set of system states and a set of transactions. For evaluation, each test condition was set and the test transactions were executed on both simulator programs and identical types of metric data were collected from each program, for analysis.

A case-by-case study was performed on the data from each of the simulation models. The details of this study is reported in Appendix C. For each test case, Appendix C defines the test conditions, provides the data from both simulation programs, and discusses the test case results.

5.7.3 Software System Structures

The application software included the hierarchy of modules required to meet the system specification and the control logic necessary to drive the computation flow and detect and recover from selected faults. The modules executed the control logic necessary to allow an execution flow but the modules did not contain actual processing logic (i.e., tracking algorithms or graphic software, etc.). An execution of a transaction by one of the simulation models represents a traversal of the computation graph.

Fault management was explicitly modeled. In the case of the object structure, software faults, within a type manager can only affect objects of that type. In the case of the functional layered structure, all of the data which can be reached in a particular layer could be destroyed.

5.8 Execution

In each case the execution of the simulation model was accomplished by defining the paths through the computation graph defined for each of the system methodologies and then traversing the graphs summing the metric values. The vertices of this dependency graph are the functions of the type managers in the case of the object-oriented system, and the functions embedded in the layers in the case of the functional system. Thus, evaluation by computation graph traversal allows one to gather data about each of the methods which we have defined. Each procedure call implies a context switch. When the procedure call was across type managers or across processes there is a larger cost than for internal (i.e., internal to a type manager or process) context switching. The flow of data between modules is measured by evaluating the data carried on each arc; inter-module transfers of data can be distinguished from an inter-type manager or inter-process flow of data.

5.9 Metric Data Description

This section will give a description of the specific metric sums that were kept by each of the simulation programs during the execution of a test case.

CPU Context Switches is the performance cost metric sum that was associated with changes in any computer execution domain during the processing of a transaction. The context switch metric was set to one for a computer subroutine call. The context switch metric was set to 10 for a message received by a process (in the functional system) or by a type manager

(in the object system) from a process, from a type manager, or from a communications, disk, or display hardware unit. This function was chosen because message context switching generally has a much higher performance cost in real systems. Within the simulation programs, the CPU context switch metric was set to zero for processing within any of the non computer units.

CPU 1 messages, *CPU 2 messages*, and *CPU 3 messages* are a special class of context switches. They are measures of the number of messages sent to each respective CPU during the execution of a test case. These metric sums are separate and distinct from the CPU context switches sum. The respective metric sum was increased by one whenever a message was sent to computer 1, 2, or 3.

Proc/TM switches are a special class of context switches. For the functional system this is a count of the number of times that a process domain was entered and for the object system the number of times that a type manager domain was entered. These metric sums are separate and distinct from the CPU context switches sum.

CPU Control Flow and *CPU Data Flow* are the performance cost metric sums that were associated with the passing of control information and data into any computer execution domain. For each computer subroutine, process, or type manager invocation, processing domain-specific metric values were added to the respective sums.

CPU 1 data, *CPU 2 data*, and *CPU 3 data* are special cases of the control and/or data flow metric sums. They are measures of the number of control information and/or data words sent to each respective CPU, on the local area network, during the execution of a test case.

Link data flow is a measure of the number of control and/or data words passed on both of the system local area network, among hardware units.

Disk data flow is a measure of the amount of control information and/or data received by both of the system disk units.

Display data flow is a measure of the number of words of control information and/or data received by both of the operator display units.

Chapter 6

METHODOLOGY ANALYSIS

This chapter will first give a quantitative analysis of the design methodologies based upon the experiments described in chapter 5 and reported in detail in Appendix C. Then a qualitative analysis of the design methodologies will be given based upon the inherent properties of the methodologies. Finally, the author's conclusions will be presented.

6.1 Quantitative Analysis

The context switches, CPU control flow, and CPU data flow were the primary metrics selected for system evaluation. The primary metrics were performance measures of the workload resulting from the respective software system structures. Other secondary metrics were collected and will be used, as needed, to help explain differences in the primary metrics. The secondary metrics were considered components of the workload resulting from the respective software structures.

Execution of the object system yielded from 33 to 59 percent fewer context switches than execution of the functional system. Within the range, the difference was lower during cases that require system configuration (i.e. cases 1, 4, 6, 7, 9, 10, and 12) and higher during cases that were exclusively

message processing (i.e., cases 3, 5, 8, and 11). See Appendix C for the case results and discussion of the case results.

The control flow metric sums were consistently (i.e., cases 2 through 12 were between 19 and 30 percent) higher for the functional system. In case 1 the functional system's control flow metric sum was even higher (452 percent) because case 1 consisted exclusively of control sequences.

The object system's data flow metric sums showed a typical four or five percent improvement over the functional system's metric sums. In case 1 the functional system's data flow metric sum was higher relative to the object system but the numbers were very small because data processing did not occur. There was a large percentage difference in data flow due to the difference in how the respective systems handled display initialization data.

The display data flow and disk data flow metric sums were always the same between systems and, except for case 1, always the same between cases. Case 1 differed because it was the system initialization case. For each case these results confirm that the same application task was always being done by both systems. For all cases link data flow was similar between systems but CPU 1, 2, 3 messages and CPU 1, 2, and 3 data varied greatly between systems.

One might intuitively expect in comparing a layered functional system to a traditional (i.e., non-vertically partitioned) object system design:

(1) a higher context switching in the object case because of the greater granularity of the computation domains; (2) equivalent or increased data flow in the object design due to the passing of data to layered general purpose system wide processing routines to perform operations on the data; (3) equivalent or higher control flow, because in passing data, control information to command operations on the data also needs to be passed.

In the experiments the observed performance advantage of the vertically partitioned object design was predominantly due to: (1) the type managers performing all operations on their own data, (2) there was no operating system and hence, no operating system process switching, (3) inter-type manager data access synchronization was not required because each type manager had exclusive control of its data.

Because type managers perform all operations on their data there was a reduction in data flow. In the experiments, the functional system passed output data to an operating system routine that performed the output. The object system requested access to the network and performed its own output. Hence, the object system did not incur a data flow penalty. This effect was best observed in the results of case 1, but was a factor in the data flow results of all the cases.

Operating system process switching was a significant performance cost contributor in the functional system. In the object system, messages were sent directly between type managers. The distributed switchboard type

manager provided routing information to local type managers, as was needed. In the functional system all inter-process messages first were decoded by the operating system and then passed to the appropriate process. This affected the context switches, data flow, and control flow metric sums. Another significant contributor to the control flow metric was the fact that an inter-process context switch had a higher control flow cost than an inter-type manager context switch. This occurred because a type manager context switch required less passing of control information than a process switch. The effect of operating system process switching was most evident in case 1 but was a contributing factor to the results of all the cases.

In the object system inter-type manager data access synchronization was not needed because type managers are solely responsible for storage of the occurrences of their data object types. Other type managers do not have access to their data. Consequently, file locking and unlocking was not needed when data was being written by the object system. In the functional system, as a part of the disk data write protocol, the writing CPU had to send both file lock and file unlock messages to the other CPUs. The effect of this design difference can most easily be seen in case 3, but was a contributing factor to the results of all the cases except case 1 which did not have disk write operations. The functional system's file lock/unlock protocol had a significant impact on the context switching and control flow metrics because file lock/unlock messages had to be constructed, sent, received, decoded, and processed.

6.2 Qualitative Analysis

This section will provide a qualitative analysis of the structure of each system.

The motivation for an object-oriented system designs is increased comprehensibility and reduced software complexity. It has been "folklore" that these virtues can be bought only at the cost of efficiency. A vertically partitioned object program structure was shown to provide a mechanism by which we may compose elementary (primitive) operations on primitive data objects to form logical data structures and logical operations which retain efficiency.

Many fault recovery concepts discussed in chapter 2, especially from transaction-oriented software systems, can be transformed to the object-oriented structure proposed here (discussed in chapter 3). The vertically partitioned object design approach has been shown to avoid many of the complications in fault recovery, discussed in chapter 2. These complications arise when transactions are used to provide atomicity.

In chapter 3, it was shown that the top down hierarchy of composite objects (i.e., composed of objects) provides a simple means for fault recovery. An important concept is that a hierarchical structure must be maintained within composite objects. In a typical layered hierarchical system a hierarchical view is taken of the entire system, whereas the vertical partition method results in a hierarchical view of each object type.

Chapter 2 concluded that the concept needed to provide efficient and manageable fault recovery gleaned from the literature was that actions should be composed into atomic steps that meet invariant conditions before entering and/or leaving the object type manager's boundaries. Atomicity must be maintained within each type manager's domain. In the object-oriented design approach, when a fault is detected, each object's type manager may either correct or compensate for the fault, thereby masking the fault or signaling to a higher level object to possibly act on the fault indication. As such, hierarchical vertical partitioning at the object level provides a limitation on fault propagation.

Well designed type managers will minimize the flow of data between object modules because they are designed around their data structures and all operations on the data are done locally. The resultant passing of controls and names helps system performance and is a very good security feature.

The choice of recovery mechanisms within each type manager is an important factor with regard to efficiency and performance. Chapter 3 discussed how the semantics of the object can be used to allow programmed exception handling resulting in compile time binding. Efficiency is enhanced by compile time binding of the sequence of elementary operations to a set of data items.

The classical objection to object-oriented design is that they are not efficient. The results given here suggest that this lack of efficiency is due to

building objects on top of a traditional layered structure. Vertical partitioning within the objects encourages the use of the semantics by the type manager of each object. The keys to the performance improvement are type managers designed around data structures, compile time binding of functions to data, and the ability of the type managers to use the object's semantic properties to optimize data processing, execution control, fault location, and recovery.

It was the author's observation that the object program was easier to code, modify, and debug because changes or problems could generally be localized to one or more specific type managers. Control logic changes and problems in the functional system typically involved complex interactions between the operating system and the application program.

6.3 Conclusion

The results of the methodology evaluation experiments proved that a vertically partitioned object program can have competitive performance efficiency with a layered functional program. Performance efficiency was quantified by the software metrics of context switches, CPU control flow, and CPU data flow.

It is the author's belief that the design approach proposed in this thesis can result in software that is an improvement over previous attempts to balance dependability and performance while maintaining high software comprehensibility and low software complexity.

Appendix A.

EXPANDED APPLICATION ENVIRONMENT

A.1 Introduction

This appendix will describe a set of requirements for an application system that is of interest to the author. These requirements were the basis for the simplified set of requirements that are given in chapter 4. Much of this material is based upon [Tripathi 83].

A.1.1 Execution Environment

The execution environment that will be considered is a "soft" real-time environment, where soft implies that processing and responses must happen in near real-time. The application programs will support data collection, communications, data processing and decision support, control of external devices and/or systems, and user interfaces. Distributed concurrent processing must be supported using local and global communications networks (e.g., both high and low bandwidth channels) on a diversity of processors. The requirements for this environment will now be reviewed.

A.1.2 System Functions

The general goals for a soft real-time data processing system are the following.

- Acquire sensor data from peripheral devices and/or systems.
- Acquire operator control information.
- Provide operator decision support.
- Make information available to the operator.
- Provide priority response time to time-sensitive operations.
- Support the local and global database needs of the system.
- Provide a reliable and prioritized (for time sensitive processes) message facility between processes and processors.
- Provide an automated degraded mode operating capability (avoid single points of failure).
- Support multi-user multi-level security of information.

A.1.3 Operational Environment

Because of the evolutionary nature of digital systems, it is desirable to adopt a modular design approach which allows changes to the system for expansions, capacity upgrades, functional upgrades, hardware substitutions, and additions of new elements. Each replaceable module type within the system must externally meet the same functional specifications. As an example, processors must support a common high order language (i.e., ADA) with virtual interface handlers. The approach of modular system design should also help in rapidly configuring new systems.

Instead of designing systems to meet certain specific requirements, it is desirable to provide an architecture which can easily adapt to the long-term changing requirements due to the state of the technology and changing short-term and long-term applications.

A.1.3.1. Physical Environment

A system will consist of clusters of local nodes that have high bandwidth (e.g., 1-10 mb/sec) intra-node communications (e.g., local area networks within the clusters) that are interconnected by lower bandwidth (e.g., less than 10 kb/sec) inter-cluster communications (e.g., global networks between the clusters). Each node may have sensors, secondary storage, other peripheral devices, and user interfaces. Most of the intra-cluster messages consist of remote procedure calls, database updates, and query messages. Most of the inter-cluster communications consists of command messages and database updates and query messages.

The following assumptions are made.

- The system is constructed of unreliable processors, unreliable secondary storage, and unreliable communications.
- There is sufficient hardware redundancy and protocols for the utilization of this redundancy to eliminate any single points of failure.
- There is sufficient hardware redundancy to allow reasonable fault recovery.

The system must provide:

- stable processors, stable storage, stable communications;

- the ability to reconstruct the database from the replicated components of the global database;
- the ability to have safe recovery from software and hardware faults.

In summary, the system must accommodate network partitioning, node dropout, node reunion, and network reconfiguration.

A.2 System Requirements

The system will be a distributed soft real-time vertically structured object oriented system. The following functional system goals must be considered.

- Directory services for users, software modules, and data.
- Configuration management.
- Fault tolerance in the event of faults.
- Allocation of shared resources.
- Inter processor communications.
- Relocation of software programs.
- Multi-level data security, access control, release control, and audit trail.
- Processor load leveling, by task relocation and/or parallel operations, to maximize overall system throughput.
- Access to global system software.
- Inter processor communications.

- Performance monitoring.

A.2.1 Distributed System Subfunctions

The subfunctions of interprocess communications, resource management, security, configuration management, and database management will be further specified.

A.2.1.1. Interprocess Communications

Interprocess communications between local and global nodes need to be supported for the following functions.

- Database query and updates between nodes.
- Remote invocation of functions between processors.
- Status query and other status management functions between distributed operating system functions.
- System recovery after failure.
- Message passing.

A.2.1.2. Resource Management

Resource management functions of the distributed system include maintaining directories for the resources distributed over the network. The resources managed by the distributed system include data base objects, system service processes, processors, I/O devices, and secondary storage devices.

The resource management functions include the following.

- Location transparency in accessing the resources.
- Location transparency in the execution of software programs.

- Uniform mechanisms for accessing resources of different types.
- Prioritized scheduling of tasks and allocation of resources such as processors to tasks.
- Handling of local and external service requests from within and external to each processor.
- Detection and handling of certain error conditions.
- Resolving deadlock.
- Concurrent operations on shared resources.
- Protection among users and user tasks.
- Name management.
- Directory management for distributed resources.

A.2.1.3. Security

The distributed type managers are concerned with the transfer of information between processes and objects. The following may be provided.

- Authentication between processes.
- Cryptographic transfer of information, when required, which may potentially include dynamically varying security cryptography for highly sensitive data.

A.2.1.4. Configuration Management

Configuration management entails the continuous and efficient reconfiguration of the system's processing, storage, and communication resources to accommodate faults, dynamic variation in the workload, and performance optimization. Configuration management activities include the following.

- Node addition and secession from system network.
- Reallocation of data elements between nodes.
- Location of software program units.
- Allocation of backup responsibilities to cells.
- Maintenance of resource directories.
- Node status management.
- Interconnection (routing tables) management.
- Backup allocation of processors to functions.
- Database to storage device mapping.
- Test and diagnosis of nodes.

A.2.1.5. Database Management

Varying degrees of mutual consistency across multiple copies of an object at more than one node does not have to be absolute; i.e., the data may be hours old and still be acceptable as long as the age of the information is known. The consistency requirements will vary between objects. This type of data redundancy requires the consideration of different recovery for different types of data.

The system database functions should address the following areas:

- Creation of new data types.
- Creation and deletion of database objects.
- Database partitioning and replication.

- Dynamic relocation of database objects.
- Uniform database interfaces.
- Probabilistic algorithms to address the consistency issue.
- Error detection and handling of node failures, communications failures, system reconfiguration.
- Query decomposition.

A.3 Assumptions

The following assumptions are being made in the development of this work.

- There is no logical sharing of memory or devices between objects. There can be sharing of logical resources within an object or between component objects of an object (note that an object may be composed of component objects).
- Distributed computing will be supported and is user transparent. All objects in the system are accessed in a uniform fashion regardless of their location.
- Varying degrees of concurrency will be supported and will be controlled by the individual type managers.
- "Remote procedure calls" with "at most one execution" semantics will be supported (e.g., all calls and passing of data outside of an object's boundaries are done by procedure calls). [Nelson 81].
- The underlying machine and/or kernel supports reliable message passing. Messages will be passed in the order that they are sent.
- The underlying machine is "Fail Stop", that is, the machine stops computation and provides an indication when a failure occurs.

Appendix B.

METRIC LITERATURE REVIEW

B.1 Introduction

Software metrics are developed and used to quantify and evaluate software. In this application, we need metrics that can be used for quantitative evaluation of the utility value of knowing and using the semantics of the functions which are being executed against the data. The metrics should have the option of specificity, within object type managers, of the mechanisms used for recovery and consistency control partitioning at the object boundaries, that limits the impact of a given function and/or a given fault.

The software metrics requirement is to generate meaningful metrics for the enhancements in reliability which will come from each of these properties.

B.2 Software Metrics

There is no well established set of rules or concepts for analyzing or evaluating the properties of software systems. This is because the variables which can be measured do not map readily upon properties or characteristics which are both quantifiable and comparable across programs. Only properties or characteristics which are both quantifiable and comparable across programs

can be realistically used as metrics for evaluation of software systems. Since there is not a uniform basis for evaluating the software we develop, it is difficult to make meaningful evaluations or comparisons of the methodologies used to produce the software programs. Quantification is the process that enables us to find the relationships between concepts, that allows us to transform an art or a craft into a science. Fundamental properties that may be useful metrics are identified by their appearance in the invariant relations which unite observations across many environments; these observations commonly express conservation relations. These fundamental properties may be derived from the invariant principles which underlie experimental observations [Browne 81].

Some of the software properties that have been cited in the literature as meaningful software metrics are the following.

- Syntactic measures [Gilb 77] [Halstead 77] [Thayer 76].
- Count of modules and module connections [Belady 79], [Gilb 77] - this gives a measure of the graph complexity.
- Program control flow [McCabe 76] [Ryder 79].
- Program data flow [Myers 78] - a good design will try to minimize the flow of data between modules. In the object oriented approach the operations should be done locally by the TMs; all work should be done locally and only commands and names should be passed.
- Amount of information shared between modules [Chanon 73] - this measure is a composite of the program control flow and the program data flow.
- Program complexity - this can be based on a count of the number of predicates [McCabe 76] or based on the functions (operators) and operands within a program level or boundary.

- Software reliability [Walters 78] - this can be shown for a particular application, by doing the dependency graph. If a function, module, or device fails it is desirable to know what percent of the data is lost, what the propagation of the fault is, and what dependencies are associated with a particular fault. Dependency graphs can be compared but are hard to quantify.

- Response time - this measure is only a good comparison between various implementations of the same system.

- Resource Consumption [Lynch 81] - useful when an externally defined unit of work is defined.

- Readability [Schneiderman 80].

- Brevity [Schneiderman 80].

- Correctness - must be done by proof of correctness. The arguments are qualitative.

- Reliability [Walters 78] - you need to show fault locality, the amount of work to detect errors, element fault diagnosability (due to semantics), the amount of work to detect errors, and that semantics allow better error detection in order to demonstrate improvements in reliability. This can only be quantified by example.

- Efficiency and performance [Walters 78].

- Integrity [Walters 78].

- Usability [Walters 78].

- Maintainability [McCall 77].

- Testability [McCall 77].

- Modifiability [McCall 77].

- Portability [McCall 77].

- Reusability [McCall 77].

- Interoperability [McCall 77].

- Length of recovery path.

- Execution cost at a given level of integrity.

- Recovery cost.

As in software design, there should be a hierarchical structure in metrics, so that comparable properties may be compared at equivalent levels of abstraction.

The quantifiable metrics that will primarily be used for the evaluation of this study are the following.

- The dependency graph showing the functional interconnection of modules as a result of vertical partitioning and the impact of faults upon the system. These dependencies can be quantified by the number and length of the recovery paths for each module in the system.

- The amount of data passed between modules (as shown on the arcs of the dependency graph.)

- The number of control words and names passed per arc of the dependency graph.

The other metrics listed above will be subjectively talked about, as appropriate.

B.2.1 Complexity Measure

The applicable software complexity metrics of Halstead will be developed in this section because they frequently are used to quantify software complexity. The following metrics will be developed:

- program volume,
- program length,
- program level, and
- intelligence factor.

The following definitions will be used.

- n1 is the number of unique or distinct operators appearing in the implementation.
- n2 is the number of unique or distinct operands appearing in the implementations.
- N1 is the total usage of all the operators appearing in the implementation.
- N2 is the total usage of all the operands appearing in the implementation.
- f(1,j) is the number of occurrences of the jth most frequently occurring operator, where j = 1,2,..., n1.
- f(2,j) is the number of occurrences of the jth most frequently used operand, where j = 1,2,...,n2.
- n = n1 + n2 is defined as the vocabulary.
- N = N1 + N2 is defined as the implementation length.

The following metrics are developed by Halstead:

- $NN = n1 \log2 n1 + n2 \log2 n2$, where NN is the calculated length of an implementation of an algorithm in terms of its vocabulary approximated by taking the power set (the set of all possible subsets) of the number of operators and operands: 2**N = n1**n1 * n2**n2 and solving for N. NN is used in place of N to distinguish the quantity obtained, the calculated length, from the value of the length N obtained by direct observation.

- $V = N \log2 n$, where V is the volume or size of an implementation. N is the length (or N1 + N2) and n is its vocabulary (or n1 + n2). This metric should be sensitive to when a program is translated from one language to another, since in any one language some algorithms are smaller than others. Consequently, V is a measure of the relative power of a representation language. To be independent of the character set used this metric has binary digits or bits as the units.

- $V^* = (2 + n2^*) \log2 (2 + n2^*)$, where V* is the minimal or potential volume that an algorithm can be implemented. V* is the most succinct form in which an algorithm could ever be expressed, in a language where all operations are already defined (possibly as a subroutine or procedure). V* is a measure of an algorithm's content and V* of any algorithm should be independent of any language in which it might be expressed. To start $V^* = (N1^* + N2^*) \log2 (n1^* + n2^*)$, where in its minimal form neither operators nor operands require repetition; thus $N1^* = n1^*$ and $N2^* = n2^*$, where $n1^* = 2$ (one distinct operator for the naming of the function or procedure and another to serve as an assignment or grouping symbol). In this representation, n2*, for small algorithms, should represent the number of input/output parameters and represents the number of conceptually unique operands involved.

- $L = V^*/V$, where L is the program level of the implementation of an algorithm. From this it follows that only the most succinct expression possible for an algorithm can have a level of unity. This representation (where $V^* = L * V$) also suggests that, for a given algorithm, when volume goes up, the program level goes down. From this point of view, a language with L=1 is possible when all

functionality is implemented as procedure calls. Human factors studies have indicated that the difficulty of comprehension varies inversely with the program level, assuming that the person is fluent in the chosen language.

- $LL = n1^*/n1 * n2/N2$ where LL is a measured approximation to the program level. This derivation comes from the approximation of $L = n1^*/n1$ and $L = n2/N2$. Combining these approximations and noting that the constant of proportionality must be unity gives the above equation for LL. LL is used in place of L to indicate that the level measured by LL is an approximation to the value L.

- $I = LL * V$, where I is the intelligence content. This relationship indicates that for a given algorithm, the product of volume times level should be constant as that algorithm is expressed in different languages. This value of I represents the fundamental property of information content of the algorithm.

- $I = 2/n1 * n2/N2 * (N1 + N2) \log_2 (n1 + n2)$, by substitution the equations for LL and V is in $I = LL * V$. Here all the terms on the right hand side are directly measurable from any expression of the algorithm. It should be noted that I directly correlates with V^*. Both of these metrics are independent of the language in which an algorithm is expressed.

It should be the goal of a software design effort to have resultant software that has an ideal program level ($L=1$) for a complex algorithm (high intelligence level I and potential volume V^*).

Appendix C.

EXPERIMENTAL RESULTS

This appendix contains the results obtained from the methodology evaluation experiments. The design, implementation, and execution of the experiments was presented in chapter 5. For each test case we will present (1) the initial conditions, (2) selected metric data sums (that were defined in chapter 5) from both the functional and object programs and a percentage difference ("Functional Program data"/"Object Program data - 100") of the individual metric data sums, and (3) a discussion of the test case's results. An overall experimental analysis was presented in chapter 6. It should be noted that each test case systems' (i.e., functional system and object system) conditions are cumulative. That is, the resultant condition of the preceding case plus any defined changes become the initial conditions for the current case. Throughout the discussions, reference will be made to "both systems" or "each system". Systems refers to the functionally designed software system and the vertically partitioned object designed software system, each system running on an identical simulated hardware test bed.

The context switches, CPU control flow, and CPU data flow were the primary metrics selected for system evaluation. Other metrics were collected and will be used, as needed, to help explain differences in the primary metrics.

C.1 CASE 1: Cold Startup

INITIAL CONDITION: All of the system hardware units were operational, but the systems were in standby (i.e., the three computer processing units (CPUs) were not loaded with their application software and neither communications unit was receiving external messages). A sequence of messages were put on local area network (LAN) 1, by the system control unit; the messages commanded each hardware unit (i.e. communications 1 & 2, display 1 & 2, disk 1 & 2, and computers 1, 2, & 3) to initialize and begin running.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	263	188	39
CPU Control Flow	:	2234	428	421
CPU Data Flow	:	8616	36	2293
Link Data Flow	:	8320	8140	2
Disk Data Flow	:	20	20	0
Display Data Flow	:	8000	8000	0
CPU 1 Messages	:	3	1	200
CPU 2 Messages	:	3	1	200
CPU 3 Messages	:	3	1	200
CPU 1 Data Flow	:	70	10	600
CPU 2 Data Flow	:	70	10	600
CPU 3 Data Flow	:	70	10	600
Proc/TM Switches	:	18	17	-22

DISCUSSION: Each unit received its initialization message and did the following: 1) the communications units 1 and 2 each flushed their buffers and were enabled to accept and maintain a circular buffer of the most recent ten external messages. 2) Both display 1 and 2 units erased their respective operator displays and enabled themselves to receive display-commands messages. 3) Disk 1 and 2 each initialized their interface to accept commands.

4) Computers 1, 2, and 3 each loaded and initialized their software for their respective processing responsibilities.

Computer 1 was configured to do communications processing. CPU 1 first, enabled communications unit 1 as active. Communications unit 1 was thereby enabled to accept externally received track-update messages and reformat them for local area network (LAN) transmission and to put the resultant track-update message on LAN 1, addressed to CPU 1. CPU 1 was enabled to accept track-update messages, to write a copy of the track-update message on both disks, to do track-update message decoding, to create a track-report message, and to pass the track-report message to the track processing unit. In the functional system CPU 1 also sent a status-information message to CPU 2 and to CPU 3.

CPU 2 was configured to do track processing. That is, it was enabled to receive track-report messages from the communications processor, to decode and process the track-reports, to create a display-update message, and to put the display-update message on LAN 1, addressed to the display processor. In the functional system CPU 2 sent a status-information message to CPU 1 and to CPU 3.

CPU 3 was configured to do the display processing function. CPU 3 first, sent a display-initialization message containing 4000 words of display initialization commands to display unit 1 and to display unit 2. The display units accepted their respective message and initialized the operator display.

CPU 3 was then enabled to accept display-update messages from the track processor and to reformat each message into a sequence of display hardware commands, to create two display-commands messages, and to send a copy of both of the display-commands message to display unit 1 and display unit 2, for operator presentation. In the functional system CPU 3 sent a status-information message to CPU 1 and to CPU 2.

The results of this case are of interest because they represent the configuration performance costs of the functional system compared to those of the object system. No "data processing" was associated with this case.

Context switching was higher by 39 percent in the functional system. The object system would normally be expected to have increased context switching due to greater structural granularity. That is, the functional system configured itself with a different process (communications, track, and display) and a duplicate copy of the kernel operating system on each of the three respective CPUs. The object system configured itself with eight type managers distributed amongst the three CPUs. The type and switchboard type managers were duplicated on each CPU. The communications, name, process, and program type managers were located on CPU 1. The communications type manager, on CPU 1, contained a DBM and disk type manager. The track and display type managers were located on CPU 2 and CPU 3 respectively. Analysis revealed that the increased functional context switching resulted from the processing of inter-computer status-information messages, as will be described.

In the functional system a fixed size status array of 20 words of data was sent between CPUs whenever a change of configuration was made in any CPU. That is, the changing CPU notified the other CPUs in the system of its change (i.e., a synchronized system design). Consequently, two of each of the CPU messages were status-information messages from the other CPUs. In the object system the switchboard type manager was responsible for routing of remote messages. Each local switchboard only requests status from the other CPUs, within the system, after the local switchboards fails to correctly route a message (i.e., an asynchronous system design). Functional systems typically have a capability to support a limited number of processes. Process creation and relocation in a distributed environment is typically infrequent. Hence a fixed size status array and the synchronized statusing made sense, for the functional system's design. The object system had greater structural granularity and consequently more (type managers) to status. Hence, a variable size status array was appropriate to reduce data flow to useful information. Inherent in the object design was run time binding of objects and dynamic reconfiguration. In this environment, status information within the switchboard type manager was used when it was available and correct, to reduce data flow. An actual global status update in the object design had a greater control flow cost because there typically was more information to exchange.

The control flow metric sum was 421 percent higher in the functional system's results. The functional system's dependency graph revealed that 18 process switches of which 15 were CPU kernel operating system invocations

occurred during the execution of this case; each invocation had a control flow cost of 92 words (i.e., for a total of 1656 words). The object system had 17 type manager invocations during the execution of the same case, each type manager invocation having a control flow cost of 10 words. Consequently, it was concluded that a significant portion of the functional system's control flow was due to operating system process switching. Passing of status information by the functional system also increased its control flow metric sum.

The CPU data flow was significantly higher (2293 percent) in the functionally designed case. The functional system's dependency graphs revealed that the difference was due to the passing of data to operating system routines for input and output. Specifically, the significant contributor was 4000 words of display initialization data sent from the display process on CPU 3 to both of the display units (i.e., for a total of 8000 words). In the object system, type managers are responsible for their own input and output, except for synchronization among type managers, where access control is provided by the abstract machine. Hence the object system did not have the large data flow performance cost.

Disk data flow and display data flow were identical in the two designs, as might be expected. There were some minor difference in the amount of link data flow and the number of CPU 1, 2, and 3 messages and the amounts of CPU 1, 2, and 3 data flow. This was due to the differences in statusing as was described above.

C.2 CASE 2: External Messages Received

INITIAL CONDITIONS: The system was fully operational, had been initialized by case 1, and then two separate external target-update messages were received by the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1414	1035	36
CPU Control Flow	:	25700	21150	27
CPU Data Flow	:	71246	68122	4
Link Data Flow	:	5660	5612	0
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	34	29	17
CPU 2 Messages	:	6	5	20
CPU 3 Messages	:	6	4	50
CPU 1 Data Flow	:	1484	1506	-5
CPU 2 Data Flow	:	152	137	10
CPU 3 Data Flow	:	260	205	26
Proc/TM Switches	:	46	50	-8

DISCUSSION: Communications hardware unit 1 reformatted the target-update messages and individually put the reformatted target-update messages on LAN 1, addressed to CPU 1. First, CPU 1 received the track-update message and executed the disk write protocol to write a copy of the message to the input message log on both disk 1 and disk 2. CPU 1 then executed the communications message decoder function that resulted in a track-update report. In the object system CPU 1 sent a status-request message to CPU 2 and CPU 3 and received status-information messages from CPU 2 and CPU 3. The functional system had exchanged status during execution of case 1.

CPU 1 put the resultant track-update report message on LAN 1, addressed to CPU 2. CPU 2 received and decoded the message and executed the track-update function that resulted in a display-update message. In the object system CPU 2 sent a status-request message to CPU 1 and CPU 3 and received status-information messages from CPU 1 and CPU 3.

CPU 2 put the display-update message on LAN 1, addressed to CPU 3. CPU 3 received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages). Display units 1 and 2 received the messages and updated the operator displays.

Next, CPU 1 received the second track-update message and executed the disk write protocol to write a copy of the message to the input message log on both disk 1 and disk 2. CPU 1 then executed the communications message decoder function that resulted in a track-update report. CPU 1 put the resultant track-update report message on LAN 1, addressed to CPU 2. CPU 2 received and decoded the message and executed the track-update function that resulted in a display-update message. CPU 2 put the display-update message on LAN 1, addressed to CPU 3. CPU 3 received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages). Display units 1 and 2 received the messages and updated the operator displays.

In the functional system CPU 1 was sent 34 messages; (24) messages were disk replies (i.e., either read data or write-acknowledgement messages), (2) were track-update messages from communications unit 1, and (8) were file lock/unlock acknowledgement messages from CPUs 2 and 3. In the object system CPU 1 was sent 29 messages; (24) messages were disk replies, (2) were track-update messages from communications unit 1, (2) were status-information messages, one each from CPU 1 and CPU 2, and (1) was a status-request from CPU 2.

In the functional system CPU 2 was sent (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were track-report messages from CPU 1. In the object system CPU 2 was sent (5) messages; (2) were track-report messages from CPU 1, (1) was a status-request message from CPU 1, and (2) were status-information messages from CPU 1 and CPU 3 respectively.

In the functional system CPU 3 was sent (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were display-update messages from CPU 2. In the object system, CPU 2 was sent (4) messages; (2) were display-update messages from CPU 2 and (2) were status-request messages from CPU 1 and CPU 2 respectively.

The object system's results in this case may be compared to the object system's results in case 3, because during this case the object system exchanged status between CPUs when initially sending messages to remote type managers. This was because the local switchboard manager had not been

invoked during case 1. The functional system's CPUs had exchanged status information during case 1 and did not need to exchange status during this case. Hence, case 3 will be presented to provide message processing results that are comparable between system. The object system's context switches and control flow metric sums were higher in this case, because of the exchange of status-information messages during this case.

A detailed analysis will be given as a part of the more interesting case 3.

C.3 CASE 3: External Messages Received

INITIAL CONDITIONS: The system was fully operational, inter-CPU status had been passed (i.e., during case 1 for the functional system and during case 2 for the object system), and two separate external target-update messages were received by the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1414	888	59
CPU Control Flow	:	25700	20512	25
CPU Data Flow	:	71246	67894	4
Link Data Flow	:	5660	5340	5
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	34	26	30
CPU 2 Messages	:	6	2	200
CPU 3 Messages	:	6	2	200
CPU 1 Data Flow	:	1484	1404	5
CPU 2 Data Flow	:	152	32	375
CPU 3 Data Flow	:	260	140	85
Proc/TM Switches	:	46	36	27

DISCUSSION: The execution sequence of this case was similar to the sequence discussed in case 2, the difference being that the object system exchanged status information during case 2. No status information was exchanged during this case because there were no system reconfigurations preceding or during this case.

In the functional system CPU 1 was sent 34 messages; (24) messages were disk replies (i.e., either read data or write acknowledgement messages), (2) were track-update messages from communications unit 1, and (8) were file lock/unlock acknowledgements from CPUs 2 and 3. In the object system CPU

1 was sent 26 messages; (24) messages were disk replies, and (2) were track-update messages from communications unit 1.

In the functional system CPU 2 was sent (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were track-report messages from CPU 1. In the object system CPU 2 was sent (2) track-report messages from CPU 1.

In the functional system CPU 3 was sent (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were display-update messages from CPU 2. In the object system CPU 3 was sent (2) display-update messages from CPU 2.

This is an important case because it represents the fully operational steady state system computation work load. The overall results are encouraging because in all metric categories, the object system's metric sums indicate the same or lower performance costs than in the functional system. That is, the results, from this case, can be interpreted to say that, the object system, as quantified by the performance metric sums, was as good as or better than the functional system.

The object approach had less overall context switching (about 59 percent difference) within the system CPUs. The dependency graph analysis revealed that context switching was reduced in the object system design because, (1) the object system did not lock files before writing data to disk

and (2) the object system had 36 type manager calls while the function system had 46 process switches of which 40 were kernel operating system calls. A vertically partitioned object system design only requires file read/write synchronization for concurrent operations within a type manager. This is because type managers are assigned storage blocks by their abstract machine and they are solely responsible for the contents of the storage blocks. Hence, the need to maintain data consistency does not require system wide inter-type manager synchronization. Due to the file directory structure of the functional system, an inter-CPU file lockout mechanism was incorporated into the design, for disk write operations. A file lock/unlock operation requires a significant amount of context switching due to the construction of the required messages, sending of the messages, decoding of the messages, and processing of the message commands. This also explains why there were six CPU 2 and six CPU 3 messages in the functional system's results and only two respective messages in the object system's results. In both designs, there were two data transfer messages from CPU 1 to CPU 2 and two data transfer messages from CPU 2 to CPU 3. The additional messages at CPU 2 and CPU 3, in the function system, are attributable to the file lock and unlock messages from CPU 1 to CPU 2 and to CPU 3. The difference in CPU 1 Messages of 34 versus 26 for the functional system and object system respectively, is due to the four pairs of file lock command acknowledgements and file unlock acknowledgements from CPU 2 and CPU 3 to CPU 1.

The object system shows an information flow improvement (25 percent for CPU control flow, 4 percent for CPU data flow, and 5 percent for

link data flow). Control flow improvement in the object system partially results from not having the file lock/unlock protocol, discussed in the previous paragraph. Similarly to case 1, an object system's data flow improvement resulted from the type manager's direct output control of the two display-commands messages of 400 words of display hardware commands each. The predominant contributor to the control flow difference was 46 process switches by the functional system, as compared to 36 type manager invocations by the object system, with analogously performance results as was discussed in case 1.

In this case, the disk and display data flow were identical, because identical application tasks were performed. There was a system difference in CPU 1, 2, and 3 (i.e., inter-processor) data flow because of the respective difference in CPU 1, 2, and 3 message numbers, as discussed earlier.

C.4 CASE 4: CPU 1 Failure

INITIAL CONDITIONS: Prior to this case the system had been fully operational and configured as resulted from case 1. CPU 1 had failed since the processing of the last message of case 3 and was non-operational, and two external target-update messages were received by the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1282	936	36
CPU Control Flow	:	24839	20651	20
CPU Data Flow	:	71272	67882	4
Link Data Flow	:	5508	5328	3
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	1	1	0
CPU 2 Messages	:	29	25	16
CPU 3 Messages	:	7	3	133
CPU 1 Data Flow	:	110	110	0
CPU 2 Data Flow	:	1334	1294	3
CPU 3 Data Flow	:	290	150	93
Proc/TM Switches	:	41	38	7

DISCUSSION: For this case, each system (i.e., functional and object) first had to recognize the failure of CPU 1, then each system had to reconfigure itself to bypass the non-operational CPU and relocate the processing functions of the non-operational CPU. Finally, each system had to process the two target-update messages.

Communications hardware unit 1 reformatted the target-update messages and put the first reformatted message on LAN 1, addressed to CPU 1. CPU 2 was monitoring the LAN activity, had captured a copy of the

message addressed to CPU 1, and had recognized that CPU 1 had not ACKed the track-update message from the communications unit. CPU 2 reconfigured its software to add the communications processing function that had been previously assigned to CPU 1. CPU 2 sent a message to communications unit 1, commanding it to send future track-update messages directly to CPU 2. Then, CPU 2 executed the communication function on the captured message and passed a track-report message to the tracking function also on CPU 2. CPU 2 executed the tracking function and sent a display-update message to CPU 3 for display processing. CPU 3 received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages); the display units updated the operator displays.

Next, communications unit 1 sent the second track-update message to CPU 2. CPU 2 performed the communications and track processing functions and sent a display-update message to CPU 3. Finally, CPU 3 performed the display data preparation function and sent two display-commands messages to each of the two display units; the display units updated the operator displays.

In both the functional and object systems CPU 1 was sent (1) track-update message from communications unit 1. Then, the systems reconfigured and CPU 1 was excluded from the system processing.

In both systems: 1) CPU 2 captured a copy of the track-update message sent to CPU 1, 2) CPU 2 recognized the failure of CPU 1 by the

absence of a message ACK on the network, 3) CPU 2 reconfigured to perform the communications and track functions, and 4) CPU 2 processed the captured message. Then, in the functional system CPU 2 was sent (29) messages; (24) were disk read or write reply-messages, (1) was a track-update message from communications unit 1, and (4) were file lock/unlock acknowledgement-messages from CPU 3. In the object system CPU 2 was sent 25 messages; (24) messages were disk replies, (1) was a track-update message from communications unit 1.

In the functional system CPU 3 was sent (7) messages; (1) was a status message from CPU 2, (4) were file lock/unlock messages from CPU 2 and (2) were display-update messages from CPU 2. In the object system CPU 3 was sent (3) messages; (1) was an advisory message that CPU 1 was down from CPU 2 and (2) were display-update messages from CPU 2.

This is a complex case to interpret because the metric data sums include performance cost data for system reconfiguration and message data processing. The metrics sums resemble a combination of the (re)configuration case 1 and the fully operational system message processing case 3.

This case has a 36 percent context switching difference, as compared to 59 percent in case 3. In this case the functional system had a reduced file lock/unlock workload because of the non-operational CPU. This reduced workload resulted in reduced context switching and control flow metric sums as compared to the object system. Specifically, the functional system number

of messages was eight less in this case, due to file locking/unlocking in the functional system design, as discussed in case 2. This file lock/unlock protocol is considered the primary reason that the functional system had an increased context switch metric sum.

The control flow metric sums differed by 20 percent. The dependency graphs revealed that in this case there were 38 object system's type manager invocations and 41 functional system' operating system invocations. The object systems superior performance indicated by the control flow metric, is attributable to the greater control flow cost of an operating system process switch, as discussed in case 1.

The object system had a slightly reduced overall data flow of 4 percent. This is attributed to the direct output of the display-commands messages (i.e., in this case each of 400 data words), as discussed in case 1. The percentage difference is less here than in case 1 because the case 1 display-commands messages were 4000 data words, of initialization data.

The functional systems provided inter-computer statusing during the reconfiguration while the object system updated status as a part of processing the first message.

C.5 CASE 5: External Messages Received

INITIAL CONDITIONS: CPU 1 remained non-operational, the systems remained configured as resulted from case 4, and two external target update message were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1238	856	44
CPU Control Flow	:	24468	20446	19
CPU Data Flow	:	71086	67882	4
Link Data Flow	:	5468	5308	3
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	0	0	
CPU 2 Messages	:	30	26	15
CPU 3 Messages	:	6	2	200
CPU 1 Data Flow	:	0	0	
CPU 2 Data Flow	:	1444	1404	2
CPU 3 Data Flow	:	260	140	85
Proc/TM Switches	:	38	34	11

DISCUSSION: The following sequence occurred twice; once for each message. Communications unit 1 sent a track-update message to CPU 2. CPU 2 performed the communications and track processing functions and sent a display-update message to CPU 3. Finally, CPU 3 performed the display data preparation function and sent two display-commands message to each of the two display units; the display units updated the operator displays.

In both the functional and object systems no messages were sent to CPU 1. In the functional system CPU 2 was sent (30) messages; (24) were disk read or write reply-messages, (2) were track-update message from communications unit 1, and (4) were file lock/unlock acknowledgement-

messages from CPU 3. In the object system CPU 2 was sent 26 messages; (24) messages were disk replies, (2) were track-update messages from communications unit 1.

In the functional system CPU 3 was sent (6) messages; (4) were file lock/unlock messages from CPU 2 and (2) were display-update messages from CPU 2. In the object system CPU 3 was sent (2) display-update messages from CPU 2.

This is an important case because the system is processing messages and no reconfiguration occurs. For analysis, this case will now be compared to case 3.

For both systems, the context switching, CPU control flow, CPU data flow, and link data flow metric sums are reduced because there are no inter-processor messages between the communications processing and the track processing functions.

The disk data flow, display data flow, CPU 3 messages, and CPU 3 data flow metric sums are identical between cases 3 and 5, as was expected. In case 3 there were 46 process switches versus 38 in case 5. While in case 3 there were 36 type manager calls versus 34 in case 5. This inter-case differences of Process/TM function domain switching was responsible for the reduction in metric values in case 5, as compared to case 3. The functional system had some advantage, in case 5 over the functional system in case 3

because of the reduced requirement of four less file lock/unlock massages and the resultant four acknowledgement messages. The object system did not gain this advantage in case 5, because it did not lock files. Consequently, the respective differences in context switches and control flow metrics between functional and object systems were less between cases. The overall differences between the function system and the object system are consistent between the cases considering the inter-case differences of Proc/TM switches.

C.6 CASE 6: CPU 1 Recovery

INITIAL CONDITIONS: CPU 1 became operational after the processing of the last message of case 5, the systems remained configured as resulted from case 4, and two external target update message were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1530	1121	36
CPU Control Flow	:	26766	20954	27
CPU Data Flow	:	71624	68003	5
Link Data Flow	:	5774	5485	5
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	19	15	26
CPU 2 Messages	:	21	16	37
CPU 3 Messages	:	8	3	166
CPU 1 Data Flow	:	802	775	3
CPU 2 Data Flow	:	868	757	-11
CPU 3 Data Flow	:	320	169	89
Proc/TM Switches	:	55	51	7

DISCUSSION: For this case, each system (i.e., the functional and object) first had to recognize the operability of CPU 1, then each system had to reconfigure itself to include the operational CPU, relocate the communications processing functions to CPU 1, and to process the two target-update messages.

Communications hardware unit 1 reformatted the target-update messages and put the first reformatted message on LAN 1, addressed to CPU 2. CPU 1 was monitoring the LAN activity, recognized that the system was active by capturing the target-update message addressed to CPU 2. CPU 1

configured its software to add the communications processing function that had been previously assigned to CPU 2. CPU 1 sent a message to CPU 2, commanding it to discontinue communications processing. CPU 2 sent a message to communications unit 1, commanding it to stop sending track-update messages to CPU 2. CPU 1 then sent a message to communications unit 1, commanding it to send future track-update messages to CPU 1. In both systems, during the execution of the first message, when CPU 1 commanded CPU 2 to terminate communications processing, CPU 2 continued to process till completion. After completion of processing CPU 2 reconfigured itself to no longer perform the communication processing function. In the functional system CPU 1 sent a status-information message to CPU 2 and to CPU 3 and CPU 2 sent a status-information message to CPU 1 and to CPU 3. In the object system CPU 1 requested and received status from CPU 2 and from CPU 3.

CPU 2 then performed the tracking function and sent a display-update message to CPU 3 for display processing. CPU 3 received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages).

Next, communications unit 1 sent the second track-update message to CPU 1. CPU 1 performed the communications processing and sent a track-report message to CPU 2. CPU 2 received the message and performed the track processing functions and sent a display-update message to CPU 3.

Finally, CPU 3 performed the display data preparation function and sent two display-commands message to each of the two display units; the display units updated the operator displays.

In the functional system CPU 1 was sent 19 messages; (12) messages were disk replies, (1) was a track-update messages from communications unit 1, (2) were status-information messages from CPU 2 and CPU 3 respectively, and (4) were file lock/unlock acknowledgements from CPUs 2 and 3. In the object system CPU 1 was sent 15 messages; (12) messages were disk replies, (2) were status-information messages from CPU 2 and from CPU 3 respectively, and (1) was a track-update message from communications unit 1.

In the functional system CPU 2 was sent (21) messages; (12) messages were disk replies, (1) was a track-update message from communications unit 1, (1) was a drop communications process message from CPU 1, (1) was a status-information messages from CPU 1, (2) were file lock/unlock acknowledgements from CPU 1, (1) was a file unlock acknowledgement-message from CPU 1, (2) were file lock/unlock messages from CPU 1 and (1) was a track-report message from CPU 1. In the object system CPU 2 was sent 16 messages; (12) messages were disk replies, and (1) was a track-update message from communications unit 1, (1) was a status-request message (i.e., that also contained CPU 1's status) from CPU 1, (1) was a drop communications message from CPU 1, and (1) was a track-report messages from CPU 1.

In the functional system CPU 3 was sent (8) messages; (2) were file lock/unlock messages from CPU 1, (2) were file lock/unlock messages from CPU 2, (2) were status-information messages from CPU 1 and CPU 2 respectively, (1) was a display-update messages from CPU 1 and (1) was a display-update messages from CPU 2. In the object system CPU 2 was sent (3) messages; (1) was a status-request message from CPU 1 and (2) were display-update messages from CPU 2.

This is a complex case to interpret because the metric data sums include performance cost data for system reconfiguration and message data processing. The results will be compared to those of case 3 and case 4.

The context switching, CPU control flow, CPU data flow metrics sums resemble those of case 4, because both case contain reconfiguration and external-message processing. The differences are attributable to the functional system exchanging (8) file lock/unlock messages and (8) acknowledgment messages here and only (4) and (4) in case 4. This difference is due to the non-operational CPU in case 4.

There were 46 versus 55 functional system process switches in cases 3 and 6 respectively. This compares to 36 versus 51 type manager invocations in cases 3 and 6 respectively. These differences provide an explanation for the differences in context switches and CPU control flow between cases.

After completion of the processing of these two messages, the systems

were configured identically as they were at the completion of case 3. As part of the validation process, test case 4 was repeated at this point and identical results, as were reported and discussed under case 4, were obtained.

C.7 CASE 7: CPU 2 Failure

INITIAL CONDITIONS: The system had been fully operational and configured as resulted from case 6. CPU 2 had failed since the processing of the last message of case 6 and was non-operational, and two external target-update messages were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1290	960	34
CPU Control Flow	:	24827	20698	19
CPU Data Flow	:	71212	67835	4
Link Data Flow	:	5420	5279	2
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	31	28	6
CPU 2 Messages	:	1	1	0
CPU 3 Messages	:	6	2	200
CPU 1 Data Flow	:	1474	1449	1
CPU 2 Data Flow	:	30	16	87
CPU 3 Data Flow	:	152	50	204
Proc/TM Switches	:	41	40	2

DISCUSSION: During this case, both systems had to recognize the failure of CPU 2, then each system had to reconfigure itself to bypass and relocate the processing function of the non-operational CPU, and each system had to process the two target-update messages.

Communications hardware unit 1 reformatted the target-update messages and put the first reformatted message on LAN 1, addressed to CPU 1. CPU 1 executed the communications functions and placed a track-report message on LAN 1, addressed to the CPU containing the track process.

In the functional system CPU 3 captured a copy of the first file-lockout message sent to CPU 2 by CPU 1 and CPU 3 recognized that CPU 2 had not ACKed the file-lockout message from CPU 1. CPU 3, in the functional system, reconfigured its software to add the track processing function that had been previously assigned to CPU 2 and sent a status-information message to CPU 1.

In the object system CPU 3 was monitoring the LAN activity, had captured a copy of the track-report message addressed to CPU 2, and recognized that CPU 2 had not ACKed the message from CPU 1. CPU 3, in the object system, reconfigured its software to add the track processing function that had been previously assigned to CPU 2 and sent CPU 1 an information message, that CPU 2 was non-operational. CPU 1 requested status from CPU 3 and CPU 3 returned a status-information message to CPU 1.

In both systems CPU 3 executed the tracking function and sent a display-update message to the display processing function, also on CPU 3, for display processing. The display processing function received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages).

Next, communications unit 1 sent the second track-update message to CPU 1. CPU 1 performed the communications processing function and passed a track-update report message to CPU 3. CPU 3 received the message and

performed the track processing functions and passed a display-update message to the display processing function (i.e., on CPU 3). Finally, CPU 3 performed the display data preparation function and sent two display-commands messages to each of the two display units; the display units updated the operator displays.

In the functional system CPU 1 was sent 31 messages; (24) messages were disk replies (i.e., either read data or write acknowledgements), (2) were track-update messages from communications unit 1, (4) were file lock/unlock acknowledgements from CPU 3, and (1) was a status message from CPU 3. In the object system CPU 1 was sent 28 messages; (24) messages were disk replies, (2) were track-update messages from communications unit 1, (1) was an advisory message from CPU 3 that CPU 1 was down, and (1) was a status-update message from CPU 3.

In the functional system CPU 2 was sent (1) file lock message from CPU 1. In the object system CPU 2 was sent (1) track-report message from CPU 1.

In the functional system CPU 3 was sent (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were track-report messages from CPU 1. In the object system CPU 3 was sent (2) messages; (1) was a status-request message from CPU (1) and (1) was a track-report message from CPU 1. Additionally, in the object system, CPU 3 captured and processed the first track-report message that was sent to CPU 2 by CPU 1.

C.8 CASE 8: External Messages Received

INITIAL CONDITIONS: CPU 2 remained non-operational, the systems remained configured as resulted from case 7, and two external target update message were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1238	854	44
CPU Control Flow	:	24468	20426	19
CPU Data Flow	:	71086	67774	4
Link Data Flow	:	5360	5200	3
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	30	26	15
CPU 2 Messages	:	0	0	
CPU 3 Messages	:	6	2	200
CPU 1 Data Flow	:	1444	1404	2
CPU 2 Data Flow	:	0	0	
CPU 3 Data Flow	:	152	32	375
Proc/TM Switches	:	38	34	11

DISCUSSION: The following sequence occurred twice; once for each message. Communications unit 1 sent a track-update message to CPU 1. CPU 1 performed the communications processing functions and sent a track-report message to CPU 3. CPU 3 performed the track processing and display data preparation functions and sent two display-commands message to each of the two display units; the display units updated the operator displays. No inter-computer status exchanges were required because system reconfiguration did not occur.

In the functional system CPU 1 was sent 30 messages; (24) messages were disk replies (i.e., either read data or write acknowledgements), (2) were

track-update messages from communications unit 1, and (4) were file lock/unlock acknowledgements from CPU 3. In the object system CPU 1 was sent 26 messages; (24) messages were disk replies, and (2) were track-update messages from communications unit 1. In both the functional and object systems no messages were sent to CPU 2.

In the functional system CPU 3 was sent (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were track-report messages from CPU 1. In the object system CPU 3 was sent (2) track-report messages from CPU 1.

This case is important because the system is configured for steady state message processing. For analysis, this case will be compared to case 5 to discuss the differences in metric values.

The only significant differences between cases 5 and 8 are the CPU 1, 2, 3 messages and CPU 1, 2, 3 data flow metrics. They differ because of the redistribution of the processing load. The other metrics are as expected.

C.9 CASE 9: CPU 2 Recovery

INITIAL CONDITIONS: CPU 2 became operational after the processing of the last message of case 8, the systems remained configured as resulted from case 7, and two external target update message were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1506	1127	33
CPU Control Flow	:	26660	20917	27
CPU Data Flow	:	71644	68014	5
Link Data Flow	:	5770	5506	4
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	35	28	25
CPU 2 Messages	:	6	4	50
CPU 3 Messages	:	8	5	60
CPU 1 Data Flow	:	1534	1441	6
CPU 2 Data Flow	:	152	108	40
CPU 3 Data Flow	:	320	193	65
Proc/TM Switches	:	55	52	5

DISCUSSION: For this case, each system (i.e., the functional and object) first had to recognize the operability of CPU 2, then each system had to reconfigure itself to include the operational CPU, relocate the track processing functions to CPU 2, and to process the two target-update messages.

Communications hardware unit 1 reformatted the target-update messages and put the first reformatted message on LAN 1, addressed to CPU 1. CPU 2 was monitoring the LAN activity, had recognized that the system was active CPU 2 configured its software to add the track processing function that had been assigned to CPU 3. CPU 2 sent a message to CPU 3,

commanding it to discontinue track processing. In both systems CPU reconfigured its software to drop track processing. In the functional system CPU 2 sent a status-information message to CPU 1 and CPU 3 and CPU 3 sent a status-information message to CPU 1 and to CPU 2.

CPU 1 executed the communications processing function and passed a track-report message to CPU 2. In the object system the track-report message was sent to CPU 3. CPU 3's switchboard type manager located the track processing function (by requesting and receiving status from the other CPUs) and forewarded the message to CPU 2 and sent an advisory message to CPU 1 that a message had been improperly sent to CPU 3. CPU 1, in the object system, sent a status-request message to CPU 2 and to CPU 3 and they returned status information messages to CPU 1.

In both systems CPU 2 performed the tracking function and a display-update message was sent to CPU 3 for display processing. CPU 3 received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages).

Next, communications unit 1 sent the second track-update message to CPU 1. CPU 1 performed the communications processing and sent a track-update message to CPU 2. CPU 2 received the message and did the track processing functions and sent a display-update message to CPU 3. Finally, CPU 3 performed the display data preparation function and sent two display-

commands message to each of the two display units; the display units updated the operator displays.

In the functional system CPU 1 was sent 35 messages; (24) messages were disk replies (i.e., either read data or write acknowledgement messages), (1) was an advisory-message from CPU 3 that a message had been improperly sent to CPU 3, (2) were track-update messages from communications unit 1, (2) were status-information messages from CPU 2 and CPU 3 respectively, and (6) were file lock/unlock acknowledgement-messages from CPUs 2 and 3. In the object system CPU 1 was sent 28 messages; (24) messages were disk replies, (2) were status-information messages from CPU 2 and CPU 3 respectively, and (2) were track-update messages from communications unit 1.

In the functional system CPU 2 was sent (6) messages; (1) was a file lock message from CPU 1, (2) were file unlock messages from CPU 1, (1) was a status-information message from CPU 3, and (2) were track-report messages from CPU 1. In the object system CPU 2 was sent (4) messages; (2) track-report messages from CPU 1, (2) were status-information messages from CPU 1 and CPU 2 respectively.

In the functional system CPU 3 was sent (8) messages; (4) were file lock/unlock messages from CPU 1, (1) was a drop-track message from CPU (2), (1) was a status-information message from CPU 2, and (2) were display-update messages from CPU 2. In the object system CPU 3 was sent (5) messages; (1) was a misdirected track-report message from CPU 1, which was

redirected to CPU 2, (1) was a status-request message from CPU 2, (1) was a drop-track message from CPU 2, and (2) were display-update messages from CPU 2.

This is a complex case to interpret because the metric data sums include performance cost data for system fault localization, system fault reconfiguration, and message data processing. Additionally, the reconfiguration/processing sequence was different between systems.

The context switching, CPU control flow, CPU data flow metrics sums resemble those of case 6. There were 55 functional system process switches in both cases. This compares to 51 versus 52 type manager switches in cases 6 and 9 respectively. These differences provide an explanation for the differences in context switches, CPU control flow, and CPU data flow between cases. In case 6 the communications processing of the first track-update message was done in CPU 2 and for the second track-update message was done in CPU 1. Here both track-update messages were processed in CPU 1.

After completion of the processing of the two external track-update messages, the systems were configured identically as they were at the completion of case 3. As part of the validation process, test case 4 was repeated at this point and identical results, as were reported and discussed under case 4, were obtained.

C.10 CASE 10: CPU 3 Failure

INITIAL CONDITIONS: Prior to this case the system had been fully operational and configured as resulted from case 9. CPU 3 had failed since the processing of the last message of case 9 and was non-operational, and two external target-update messages were received by the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1290	990	30
CPU Control Flow	:	24827	20744	19
CPU Data Flow	:	71212	67955	4
Link Data Flow	:	5560	5419	2
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	32	28	14
CPU 2 Messages	:	7	4	75
CPU 3 Messages	:	1	1	0
CPU 1 Data Flow	:	1584	1508	5
CPU 2 Data Flow	:	182	77	136
CPU 3 Data Flow	:	130	70	85
Proc/TM Switches	:	41	42	-3

DISCUSSION: During this case, both systems had to recognize the failure of CPU 3, then each system had to reconfigure itself to bypass the non-operational CPU and relocate the processing function of the non-operational CPU, and each system had to process the two target-update messages.

Communications hardware unit 1 reformatted the track-update messages and put the first reformatted message on LAN 1, addressed to CPU 1. CPU 1 executed the communications functions and placed a track-report message on LAN 1, addressed to CPU 2.

In the functional system when CPU 1 sent the first file-lock message to CPU 3, CPU 1 was monitoring the LAN activity and recognized that CPU 3 had not ACKed the message. CPU 1 reconfigured its software to add the display processing function that was assigned to CPU 3. CPU 1, in the functional system, sent a status-information message to CPU 2.

CPU 2 performed the track update function and placed a display-update message on LAN 1, addressed to the display processor. In the functional system the message was sent to CPU 1. In the object system the message was sent to CPU 3.

In the object system CPU 1 was monitoring the LAN activity, had captured a copy of the display-update message addressed to CPU 3, and had recognized that CPU 3 had not ACKed the display-update message from CPU 2. CPU 1, in the object system, reconfigured its software to add the display processing function that had been previously assigned to CPU 3 and CPU 1 sent an advisory message to CPU 2 stating that CPU 3 was down. CPU 2 sent a status-request message to CPU 1. CPU 1 returned a status-update message to CPU 1.

In both systems, CPU 1 executed the display processing function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages).

Next, communications unit 1 sent the second track-update message to

CPU 1. CPU 1 performed the communications processing function and passed a track-report message to CPU 2. CPU 2 received the message and performed the track processing functions and sent a display-update message to the display processing function (i.e., on CPU 1). Finally, CPU 1 performed the display data preparation function and sent two display-commands messages to each of the two display units; the display units updated the operator displays.

In the functional system CPU 1 was sent 32 messages; (24) messages were disk replies (i.e., either read data or write acknowledgements), (2) were track-update messages from communications unit 1, (2) were display-update messages from CPU 2, and (4) were file lock/unlock acknowledgement messages from CPU 2. In the object system CPU 1 was sent 28 messages; (24) messages were disk replies, (2) were track-update messages from communications unit 1, (1) was a status-request(with status information) message from CPU 2, and (1) was a display-update messages from the track processing function. In the object system the first display-update message sent to the non-operational CPU 3 was captured and processed by CPU 1 too.

In the functional system CPU 2 received (7) messages; (4) were file lock/unlock messages from CPU 1, (1) was a status-information message from CPU 1, and (2) were track-report messages from CPU 1. In the object system CPU 2 received (4) message; (1) was an advisory message from CPU 1 that CPU 3 was down, (1) was a status-request message (containing CPU 1 status) from CPU 1, and (2) were track-report messages from CPU 1.

In the functional system, (1) file-lock message was sent to CPU 3 from CPU 1. In the object system there was (1) display-update message sent to CPU 3 from CPU 2.

The results of this case are comparable to cases 4 and 7. The differences in metric sums are attributable to the difference in reconfiguration sequencing and the different distribution of the processing functions.

C.11 CASE 11: External Messages Received

INITIAL CONDITIONS: CPU 3 remained non-operational, the systems remained configured as resulted from case 10, and two external target update message were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1238	884	40
CPU Control Flow	:	24468	20472	19
CPU Data Flow	:	71086	67894	4
Link Data Flow	:	5500	5340	2
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	32	28	4
CPU 2 Messages	:	6	2	200
CPU 3 Messages	:	0	0	
CPU 1 Data Flow	:	1584	1544	2
CPU 2 Data Flow	:	152	32	375
CPU 3 Data Flow	:	0	0	
Proc/TM Switches	:	38	36	5

DISCUSSION: The following sequence occurred twice; once for each message. Communications unit 1 sent a track-update message to CPU 1. CPU 1 performed the communications processing functions and sent a track-report message to CPU 2. CPU 2 performed the track processing function and sent a display-update message to CPU 1. CPU 1 received the display-update message and performed the display data preparation functions and sent two display-commands message to each of the two display units; the display units updated the operator displays. No inter-computer status exchanges were required because system reconfiguration did not occur.

In the functional system CPU 1 received 32 messages; (24) messages

were disk replies (i.e., either read data or write acknowledgements), (2) were track-update messages from communications unit 1, (2) were display-update messages from CPU 2, and (4) were file lock/unlock acknowledgements from CPU 2. In the object system CPU 1 received 28 messages; (24) messages were disk replies, (2) were track-update messages from communications unit 1, and (2) were display-update messages from CPU 2.

In the functional system CPU 2 received (6) messages; (4) were file lock/unlock messages from CPU 1 and (2) were track-report messages from CPU 1. In the object system CPU 2 received (2) track-report messages from CPU 1. In both the functional and object systems no messages were sent to CPU 3.

This case is important because the system is configured for steady state message processing. For analysis, this case may be compared to cases 5 and 8.

The only significant differences between cases 5, 8 and 11 are the CPU 1, 2, 3 messages and CPU 1, 2, 3 data flow metrics. They differ because of the redistribution of the processing load. There are two extra messages in both system's message sums for this case because CPU 2 sent the display-update messages to CPU 1. In case 5 the communications and track processing functions were both located on CPU 2, hence the track-report messages did not have to be sent on the LAN. In case 8 the track and display processing functions were both located on CPU 3; hence the display-update messages did not have to be sent on the LAN.

C.12 CASE 12: CPU 3 Recovery

INITIAL CONDITIONS: CPU 3 became operational after the processing of the last message of case 11, the systems remained configured as resulted from case 10, and two external target-update message were received by both of the communications hardware units.

RESULTS:		Functional System	Object System	Percentage Difference
Context Switches	:	1550	1133	36
CPU Control Flow	:	26986	20959	28
CPU Data Flow	:	71664	68068	4
Link Data Flow	:	5810	5580	0
Disk Data Flow	:	264	264	0
Display Data Flow	:	3200	3200	0
CPU 1 Messages	:	36	29	24
CPU 2 Messages	:	8	4	0
CPU 3 Messages	:	7	4	75
CPU 1 Data Flow	:	1544	1511	2
CPU 2 Data Flow	:	212	69	207
CPU 3 Data Flow	:	290	216	34
Proc/TM Switches	:	57	52	9

DISCUSSION: For this case, each system first had to recognize the operability of CPU 3, then each system had to reconfigure itself to include the operational CPU, had relocate the display processing functions to CPU 3, and had to process the two target-update messages.

Communications hardware unit 1 reformatted the external track-update messages and put the first reformatted message on LAN 1, addressed to CPU 3. CPU 3 was monitoring the LAN activity and recognized that the system was active. CPU 3 configured its software to add the display processing function that had been assigned to CPU 1. CPU 3 then sent a

message to CPU 1, commanding it to discontinue display processing. CPU 1 reconfigured its software to disable display processing. In the functional system CPU 3 sent a status-information message to CPU 1 and to CPU 2 and CPU 1 sent a status-information message to CPU 2 and to CPU 3.

CPU 1 executed the communications processing function and passed a track-update message to the track processor on CPU 2. The tracking function was performed and a display-update message was sent to the display processor for display processing. In the functional system this message was sent to CPU 3. In the object system this message was sent to CPU 1. In the object system CPU 1 forewarded the message to CPU 3 and sent an advisory message to CPU 2 that it had incorrectly addressed the display-update message to CPU 1. CPU 2 sent a status-request message to CPU 1 and to CPU 3.

In both systems CPU 3 received and decoded the display-update message and performed the display data preparation function and sent two display-commands messages to each of the two display units (i.e., for a total of four messages).

Next, communications unit 1 sent the second track-update message to CPU 1. CPU 1 performed the communications processing and sent a track-update message to CPU 2. CPU 2 received the message and did the track processing functions and sent a display-update message to CPU 3. Finally, CPU 3 performed the display data preparation function and sent two display-

commands message to each of the two display units; the display units updated the operator displays.

In the functional system CPU 1 was sent 36 messages; (24) messages were disk replies (i.e., either read data or write acknowledgement messages), (2) were track-update messages from communications unit 1, and (8) were file lock/unlock-acknowledgement messages from CPUs 2 and 3. In the object system CPU 1 was sent 29 messages; (24) messages were disk replies, (2) were track-update messages from communications unit 1 (1) was a drop-display processing message from CPU 3, (1) was a status-request message from CPU 3, (1) was a status-request message from CPU 2, and (1) was a display-update message from CPU 2 that was forewarded to CPU 3.

In the functional system CPU 2 was sent (8) messages; (4) were file lock/unlock messages from CPU 1, (2) were status-information messages from CPU 1 to CPU 3 respectively, and (2) were track-report messages from CPU 1. In the object system CPU 2 was sent (4) messages; (2) were track-report messages from CPU 1, (1) was a status-request (including CPU 3 status-information) from CPU 3 and (1) was an advisory message stating that the first display-update message had been incorrectly sent to CPU 1.

In the functional system CPU 3 was sent (7) messages; (4) were file lock/unlock messages from CPU 1 and (2) were display-update messages from CPU 2. In the object system CPU 3 was sent (4) messages; (2) were display-update messages from CPU 2, (1) was a status-information message from CPU

2, and (1) was a status-request message (including CPU 1 status information) from CPU 1.

This is a complex case to interpret because the metric data sums include performance cost data for system fault localization, system fault reconfiguration, and message data processing.

The results of this case are comparable to cases 6 and 9. The differences in metric sums are attributable to the difference in reconfiguration sequencing and the different distribution of the processing functions.

The context switching, CPU control flow, CPU data flow metrics sums resemble those of case 6. There were 55 versus 57 functional system process switches in cases 6 and 12 respectively. This compares to 51 versus 52 type manager switches in cases 6 and 12 respectively. These differences provide an explanation for the minor differences in context switches, CPU control flow, and CPU data flow between cases.

After completion of the processing of these two messages, the systems were configured identically as they were at the completion of case 3. As part of the validation process, test case 4 was repeated at this point and identical results, as were reported and discussed under case 4, were obtained.

Bibliography

[Belady 79] Belady, L.A., C.J. Evangelisti.
System Partitioning and its Measure.
Technical Report RC 7560 - #32643, IBM T.J. Watson Research Center, March, 1979.

[Bernstein 81] Bernstein, P.A., Nathan Goodman.
Concurrency Control In Distributed Database Systems.
ACM Computing Surveys 2(13): 185-222, June, 1981.
Contains timestamp ordering protocol.

[Brinch Hansen 73] Brinch Hansen, P.
Operating System Principles.
Prentice-Hall, Englewood Cliffs, N.J., 1973.

[Browne 81] Browne, J.C., Mary Shaw.
Toward a Scientific Basis for Software Evaluation.
In Perlis, Alan, Fredrick Sayward, Mary Shaw (editor), *Software Metrics*, chapter One. The MIT Press, 1981.

[Browne 83] Browne, James C., James E. Dutton, Vincent Fernandez, Annette Palmer, Anand R. Tripathi, and Pong-sheng Wang.
Zeus: An Object-Oriented Distributed Operating System For Reliable Applications.
Technical Report, Information Research Associates, February, 1983.
This is a good summary review of the Zeus design. Zeus is an object oriented system with a transaction layer superimposed.

[Browne 84] Browne, James C.
Formulation and Programming of Parallel Computations: A Unified View.
In *Proceedings of the XI International Conference on Parallel Processing - Chicago*, pages 624-631. IEEE, 1984.

[Browne 85] Browne, James C.
Framework For Formulation And Analysis Of Parallel Computation Structures.
In *Proceedings of HICSS*. IEEE & ACM, 1985.

[Buhr 84] Buhr, R. J. A.
System Design With Ada.
Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

[Chanon 73] Chanon, Robert N.
On a Measure of Program Structure.
Technical Report, Carnegie-Mellon University, 1973.

[Cristian 82] Cristian, Flaviu.
Exception Handling and Software Fault Tolerance.
IEEE Transactions on Computing C-31(6): 531-540, June, 1982.
Good concept paper.

[DeMarcos 78] DeMarcos, Tom.
Structured Analysis and Systems Specification.
Yourdon, Inc., New York, N.Y., 1978.

[Duncan 84] Duncan, A. G.and Hutchinson, J. S.
Communication System Design Usign Using Ada.
In *IEEE 0270-5257/84/0000/0398.* IEEE, 1984.

[Garcia 83] Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In a Distributed System.
ACM Transactions on Database Systems 8(2): 186-213, June, 1983.

[Gilb 77] Gilb, Tom.
Software Metrics.
Winthrop, 1977.

[Gray 81] Gray, Jim.
The Transaction Concept: Virtues and Limitations.
In *Proceedings on Conference of Very Large Data Bases*, pages 144-154. IEEE, 1981.
Good concept paper.

[Halstead 77] Halstead, Maurice M.
Operating and Programming Systems: Elements of Software Science.
Elsevier Computer Science Library, 1977.

[Hoare 74] Hoare, C.A.R.
Monitors:An Operating System Structuring Concept.
Commun. ACM 17(10):549-557, October, 1974.

[Kieburtz 83] Kieburtz, Richard B. and Abraham Silberschatz.
Access-Right Expressions.
ACM Transactions on Programming Languages and Systems 5(1):78-96, January, 1983.

[Laprie 85] Laprie, Jean-Claud.
Dependable Computing And Fault Tolerance: Concepts And Terminology.
In *FTCS-15*, pages 2-11. IEEE, 1985.

[Lynch 81] Lynch, W.C., J.C. Browne.
Performance Evaluation: A Software Metric Success Story.
In Perlis, Alan, Fredrick Sayward, Mary Shaw (editor), *Software Metrics*, chapter Twelve. The MIT Press, 1981.

[McCabe 76] McCabe, Thomas J.
A Complexity Measure.
IEEE Transactions on Software Engineering 4(SE-2), December, 1976.

[McCall 77] McCall, J.A., P.K. Richards, G.F. Walters.
Factors in Software Quality.
Technical Report 77CIS12, General Electric, Command and Information Systems, Sunnyvale, CA., 1977.

[Moss 81] Moss, J. and B. Elliot.
Nested Transactions: An Approach to Reliable Distributed Computing.
Technical Report, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, Massachusetts 02139, April, 1981.

[Moss 82] Moss, J. Elliot B.
Nested Transactions and Reliable Distributed Computing.
In *Proceedings on Second Conference on Reliability in Distributed Systems*, pages 33-39. IEEE, 1982.
Good concept paper.

[Myers 78] Myers, G.J.
Composite/Structured Design.
Van Nostrand Reinhold, New York, 1978.

[Nelson 81] Nelson, B.J.
Remote Procedure Call.
Technical Report CMU-CS-81-119, Carnegie-Mellon University, Department of Computer Science, May, 1981.
Also available as Xerox Report No. CSL-81-8, dtd May 81.

[Parnas 72] Parnas, D.L.
On the Criteria To Be Used In Decomposing Systems Into Modules.
Communications of The ACM 15(12), December, 1972.

[Pascoe 86] Pascoe, Geoffrey A.
Elements of Object-Oriented Programming.
BYTE :139-144, August, 1986.

[Reed 83] Reed, David P.
Implementing Atomic Actions on Decentralized Data.
ACM, Transactions on Computer Systems 1:3-23, February, 1983.

[Russel 80] Russel, David L.
State Restoration in Systems of Communicating Processes.
IEEE Transactions on Software Engineering SE-6(2): 93-194, March, 1980.
Good concept paper.

[Ryder 79] Ryder, Barbara G.
Constructing a Call Graph of a Program.
IEEE Transactions on Software Engineering 3(SE-5):216-226, May, 1979.

[Schneiderman 80] Schneiderman, B.
Software Psychology: Human Factors in Computers and Information Systems.
Winthrop Publishers, Inc., Cambridge, Mass., 1980.

[Silberschatz 77] Silberschatz, A., Kieburtz, R.B., and Bernstein, A.J.
Extending Concurrent Pascal to Allow Dynamic Resource Management.
IEEE Transactions on Software Engineering SE-3(3):210-217, May, 1977.

[Thayer 76] Thayer, T.A., et. al.
Software Reliability Study.
Technical Report RADC-TR-76-238, Rome Air Development Center, 1976.

[Tripathi 83] Tripathi, Anand R., Pong-sheng Wang.
Reliability and Consistency Management Techniques In Distributed Systems.
Technical Report, Honeywell Inc, Corporate Computer Science Center, August, 1983.
RADC Contract No. F30602-82-C-0154.

[Walters 78] Walters, Gene F., James A. McCall.
The Development of Metrics for Software Reliability and Maintainability.
In *Proceedings of the 1978 Reliability and Maintainability Symposium.* ACM, 1978.

[Wood 81] Wood, W. Graham.
A Distributed Recovery Control Protocol.
In *Proceedings FTCS-11*, pages 159-164. IEEE, 1981.
Good concept paper.

[Wulf 75] Wulf, W.A.
Overview of the Hydra Operating System.
In *The 5th Symposium on Operating System Principles*, pages 122-131. ACM, November, 1975.

[Wulf 81] Wulf, William A., Roy Levine, and Samuel P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.